C4B — Mobile Robotics

Paul Michael Newman

October 2003, Version 1.00

Contents

Topic 1

Introduction and Motivation

This set of lectures is about navigating mobile platforms or robots. This is a huge topic and in eight lectures we can only hope to undertake a brief survey. The course is an extension of the B4 estimation course covering topics such as linear and non-linear Kalman Filtering. The estimation part of the lectures is applicable to many areas of engineering not just mobile robotics. However I hope that couching the material in a robotics scenario will make the material compelling and interesting to you.

Lets begin by dispelling some myths. For the most parts when we talk about "mobile robots" we are not talking about gold coloured human-shaped walking machines [1]. Instead we are considering some-kind of platform or vehicle that moves through its environment carrying some kind of payload. Almost without exception it is the payload that is of interest - not the platform. However the vast majority of payloads require the host vehicle to navigate —to be able to parameterize its position and surroundings and plan and execute trajectories through it. Consider some typical autonomous vehicle and payloads:

- Humans in a airplane on autopilot (or car in the near-ish future. CMU NavLab project)
- Scientific equipment on a Mars lander
- Mine detectors on an autonomous underwater vehicle
- Cargo containers in a port transport vehicle
- Pathology media in a hospital deliver system

[1] although the Honda P5 humanoid is a good approximation

- Verbal descriptions in a museum tour guide
- A semi-submersible drill ship (oil recovery)
- Cameras on an aerial survey drone
- Obvious military uses (tend to be single mission only....)

All of the above require navigation. This course will hopefully give you some insight into how this can be achieved.

It is worth enumerating in general terms what makes autonomous navigation so hard. The primary reason is that the majority of mobile robots are required to work in un-engineered environments. Compare the work-spaces of a welding robot in automotive plant to one that delivers blood samples between labs in a hospital. The former operates in a highly controlled, known, time invariant (apart from the thing being built) scene. If computer vision is used as a sensor then the workspace can be lit arbitrarily well to mitigate against shadows and color ambiguity. Many industrial robots work in such well known engineered environments that very little external sensing is needed — they can do their job simply by controlling their own internal joint angles. Hospitals are a different ball-park altogether. The corridors are dynamic — filling and emptying (eventually) with people on stretchers. Even if the robot is endowed with a map of the hospital and fitted with an upward looking camera to navigate off markers on the ceiling it still has to avoid fast moving obstacles (humans) while moving purposely towards it's goal destination. The more generic case involves coping with substantial scene changes (accidental or malicious) — for example doors closing in corridors or furniture being moved. The thing then, that makes mobile robotics so challenging is uncertainty. Uncertainty is pervasive in this area and we must embrace it to make progress....

Topic 2

Introduction to Path Planning and Obstacle Avoidance

A basic requirement of a mobile autonomous vehicle is path planning. With the vehicle in an arbitrary initial position **A** we wish to issue a desired goal position **B** (including orientation) and have the vehicle execute a trajectory to reach **B** . This sounds pretty simple and we can think of several ways in which we could combine simple control laws that will get us from **A** to **B** [1] Unfortunately the waters quickly become muddied when we start talking about our other concerns:

- while executing its trajectory the vehicle must not smash into objects in the environment (especially true regarding squishy humans).
- we cannot guarantee that the vehicle in question can turn-on-the-spot and would like to be able to operate a vehicle of arbitrary shape. These are called "kinematic" constraints.
- we expect only uncertain estimates of the location of the robot and objects in the environment.

The combination of path-planning, obstacle avoidance, kinematic constraints and uncertainty makes for a very hard problem indeed — one which is still an active area of research. However we can do some interesting things if we decouple some of the issues and make some

[1] for example drive in a straight line from **A** to **B** until **B** (x,y) is reached then rotate to align with **B** (theta).

Figure 2.1: Two holonomic vehicles. The underwater vehicle (ODIN, University of Hawaii) can move in any direction irrespective of pose and the complex wheel robot PPRK (CMU) driven by a "Palm Pilot" uses wheels that allow slip parallel to their axis of rotation. A sum of translation and slip combine to achieve any motion irrespective of pose.

simplifying assumptions. We shall begin by discussing vehicle properties and categorizing them into two classes — holonomic and non-holonomic.

2.1 Holonomicity

Holonomicity is the term used to describe the locomotive properties of a vehicle with respect to its workspace. We will introduce a mathematical definition of the term shortly but we will begin by stating, in words, a definition:

> A vehicle is holonomic if the number of local degrees of freedom of movement equals the number of global degrees of freedom.

We can make this a little more clear with a few examples:

- a car is non-holonomic : the global degrees of freedom are motion in x,y and heading however locally, a car can only move forward or turn. It cannot slide sideways.(Even the turning is coupled to motion).

Figure 2.2: A commercial non-holonomic robot vehicle (Roombavac from iRobot coorporation. This vehicle can be purchased for about 100 USD - but is it's utility great enough to warrant the prices? Discuss....

- The "spherical" underwater robot (Odin) and the rolling wheel vehicle in Figure 2.1 are holonomic they can turn on the spot and translate instantaneously in any direction **without** having to rotate first.
- A train is holonomic: it can move forwards or backwards along the track which is parameterised by a single global degree of freedom — the distance along the track.
- The robot "Roombavac" (iRobot corporation) vacuum cleaner in Figure 2.1 is also non-holonomic. It can rotate in place but cannot slide in any direction — it needs to use a turn-translate-turn or turn-while-drive (like a car) paradigm to move.

It should be obvious to you that motion control for a holonomic vehicle is much easier than for a non-holonomic vehicle. If this isn't obvious consider the relative complexity of parking a car in a tight space compared to driving a vehicle that can simply slide into the space sideways (a hovercraft).

Unfortunately for us automation engineers, the vast majority of vehicles in use today (i.e. used by humans) are non-holonomic. In fact intrinsically holonomic vehicles are so rare and complex (or so simple [2]) that we shall not discuss them further.

We can now place some formalism on our notion of holonomicity. We say a vehicle whose state is parameterised by a vector $\mathbf{x}$ is non-holonomic if there exists a constraint $\mathcal{Q}$ such that

$$\mathcal{Q}(\mathbf{x}, \dot{\mathbf{x}}, \ddot{\mathbf{x}}, \cdots) = 0 \tag{2.1}$$

where the state derivatives cannot be integrated out. To illustrate such a constraint we will take the case of a front wheel steered vehicle as shown in figure 2.1

[2] path planning for a train is quite uninteresting

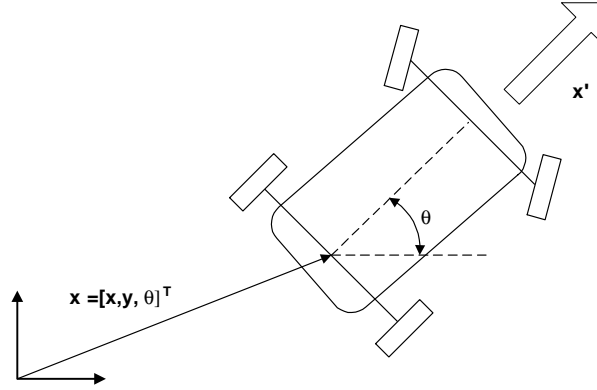



Figure 2.3: A steered non-holonomic vehicle

Immediately we can write a constraint expressing that the vehicle cannot slip sideways that is a function of **x** and its first derivative:

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} . \begin{bmatrix}-\sin\theta\\ \cos\theta\\ 0\end{bmatrix} = 0 \tag{2.2}$$

$$\mathcal{Q}(\mathbf{x}, \dot{\mathbf{x}}) = 0 \tag{2.3}$$

The state and its derivatives are inseparable and by our definition the vehicle is non-holonomic.

2.2 Configuration Space

We are describing the robot by its state $\mathbf{x} = [x_1, x_2, \cdots x_n]^T$ (for a 2D plane vehicle commonly $x_1 = x \quad x_2 = y \quad x_3 = \theta$)which is a n-state parameterisation. We call the space within which **x** resides the **configuration space** $\mathcal{C} - space$ of the vehicle:

$$\mathcal{C} = \bigcup_{\forall x_1, x_2 \cdots x_n} \mathbf{x} \tag{2.4}$$

The configuration space (or $\mathcal{C} - space$) is the set of all allowable configurations of the robot. For a simple vehicle moving on a plane the configuration space has the same dimension as the work space but for more complicated robots the dimension of the configuration space is much higher. Consider the case of the bomb disposal vehicle in figure 2.2. The configuration space for such a vehicle would be 11 — 3 for the base and another 8 for the pose of the arm and gripper. We can view obstacles as defining regions of $\mathcal{C} - space$ that are forbidden. We

Figure 2.4: A vehicle with a high dimensional configuration space - a commercial bomb-disposal platform (picture courtesy of Roboprobe Ltd. The configuration space for a human is immensely high dimensional —around 230.

can label this space as $\mathcal{C}_{\otimes}$ and the remaining accessible/permissable space as $\mathcal{C}_{\odot}$ such that $\mathcal{C} = \mathcal{C}_{\otimes} \cup \mathcal{C}_{\odot}$. It is often possible to define and describe the boundaries of $\mathcal{C}_{\otimes}$(and hence the boundaries of $\mathcal{C}_{\odot}$in which the vehicle must move) as a constraint equation. For example if the workspace of a point robot in 2D $\mathbf{x} = [xy]^T$ is bounded by a wall $ax + by + c = 0$ then

$$\mathcal{C}_{\odot} = \underbrace{\bigcup \{\mathbf{x} \mid ax + by + c > 0\}}_{\text{union of all } \mathbf{x} \text{ for which ax+by+c} > 0} \tag{2.5}$$

If each of the constraints imposed on $\mathcal{C} - space$by each obstacle k is represented by an equation of the form $\mathcal{C}_k(x) = 0$ then the open free space $\mathcal{C}_{\odot}$admitted by n obstacles can be written as the following intersection:

$$\mathcal{C}_{\odot} = \bigcap_{k=1}^{k=n} \left(\bigcup \{\mathbf{x} \mid \mathcal{C}_k(x) = 0\} \right) \tag{2.6}$$

Equation 2.6 simple states what configurations are open to the vehicle —nothing more. Any path planning algorithm must guide the vehicle along a trajectory within this space while satisfying any non-holonomic vehicle constraints and finally deliver the vehicle to the goal pose. This clearly is a hard problem. Two poses that may be adjacent to each other in state space may require an arbitrarily long path to be executed to transition between them. Take, for example, the seemingly simple task of turning a vehicle through 180^o when it is near a wall. One solution trajectory to the problem is shown in figure 2.2. The non-holonomic vehicle constraints conspire with the holonomic constraint imposed by the wall to require a complicated solution trajectory.

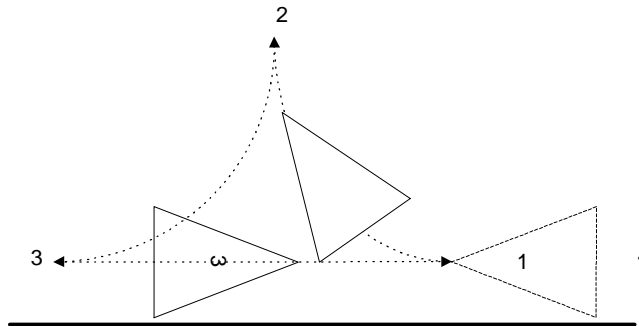



Figure 2.5: A simple task - turning through 180^o quickly becomes complicated by $\mathcal{C} - space$constraints.

2.3 The Minkowski-Sum

Real robots have arbitrary shapes and these shapes make for complicated interactions with obstacles which we would like to simplify. One way to do this is to transform the problem to one in which the robot can be considered as a point-object and a technique called the "Minkowski-Sum" does just this. The basic idea is to artificially inflate the extent of obstacles to accommodate the worst-case pose of the robot in close proximity. This is easiest to understand with a diagram shown in Figure 2.3. The idea is to replace each object with a virtual object that is the union of all poses of the vehicle that touch the obstacle. Figure 2.3 has taken a conservative approach and "replaced" a triangular vehicle with a surrounding circle. The minimal Minkowski-Sum would be the union of the obstacle and all vehicle poses with the vehicle nose just touching it boundary. With the obstacles suitably inflated the vehicle can be thought of a point-object and we have a guarantee that as long as it keeps to the new, shrunken, free space it cannot hit an object [3]. Note it is usual to fit a polygonal hull around the results of the Minkowski-Sum calculation to make ensuing path planning calculations easier.

So now we have a method by which we can calculate $\mathcal{C}_{\odot}$. The next big question is how exactly do we plan a path through it? How do we get from an initial pose to a goal pose? We will consider three methods : Voronoi, "Bug" and Potential methods.

[3]This doesn't mean that awkward things won't happen — the situation shown in figure 2.2 is still possible. However progress can be made by planning motion such that at object boundaries the vehicle is always capable of moving tangentially to the boundaries

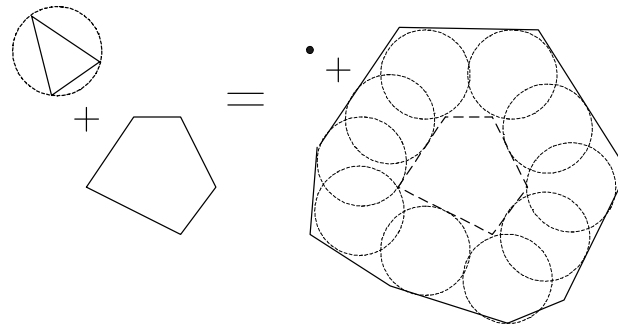



Figure 2.6: The Minkowski-Sum transforms a arbitrarily shaped vehicle to a point while inflating the obstacle. The result is guaranteed free space outside the inflated object boundary.

2.4 Voronoi Methods

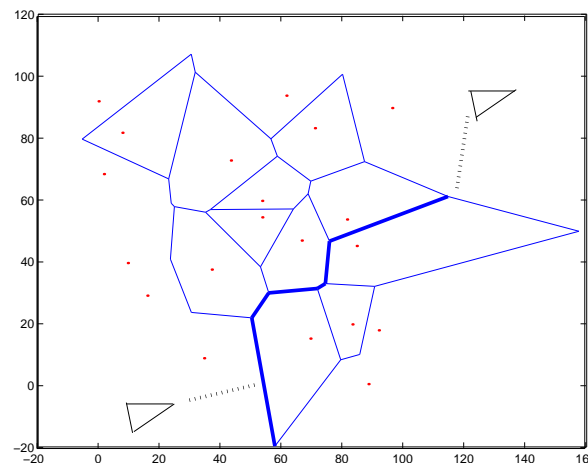



Figure 2.7: A Voronoi diagram for obstacle avoidance in the presence of point objects.

Voronoi diagrams are elegant geometric constructions [4] that find applications throughout computer science — one is shown in figure 2.4. Points on a 2D-Voronoi diagram are equi-distant from the nearest two objects in the real world. So the Voronoi diagram of two points a, b is the line bisecting them — all points on that line are equidistant from a, b. The efficient computation of Voronoi diagrams can be quite a complex matter but what is important here is that algorithms exist that when given a set of polygonal objects can generate the appropriate equi-distant loci. So how does this help us with path planning? Well, if we follow the paths defined by a Voronoi diagram we are guaranteed to stay maximally far away from

[4] *http://www.voronoi.com/*

nearby objects. We can search the set of points on the diagram to find points that are closest to and visible from the start and goal positions. We initially drive to the "highway entry" point, follow the "Voronoi - highways" until we reach the "exit point" where we leave the highway and drive directly towards the goal point.

2.5 Bug Methods

The generation of a global Voronoi diagram requires upfront knowledge of the environment. In many cases this is unrealistic. Also Voronoi planners by definition keep the vehicle as far away from objects as possible - this can have two side effects. Firstly the robot may be using the objects to localise and moving away from them makes them harder to sense. Secondly the paths generated from a Voronoi planner can be extremely long and far from the shortest path (try playing with the Matlab Voronoi function). An alternative family of approaches go under the name of " bug algorithms". The basic algorithm is simple:

1. starting from **A** and given the coordinates of a goal pose **B** draw a line **AB** (it may pass through obstacles that are known or as yet unknown)
2. move along this line until either the goal is reached or an obstacle is hit.
3. on hitting an obstacle circumnavigate its perimeter until **AB** is met
4. goto 2

In contrast to the Voronoi approach this method keeps the vehicle as close to the obstacles as possible (but we won't hit them as the obstacles have been modified by the Minkowski sum!). However the path length could be stupidly long. A smarter modification would be to replace step 3 in the original algorithm with *"on hitting and obstacle circumnavigate it's perimeter until* **AB** *is met or the line* **AB** *becomes visible in which case head for a point on* **AB** *closer to* **B** *"* Figure 2.5 shows the kind of trajectory this modified "VisBug" algorithm would execute. Clearly the hugging the object boundary is not always a good plan but interestingly it is guaranteed to get the robot to the goal location if it is indeed reachable.

2.6 Potential Methods

A third very popular method of path planning is a so called "Potential Method". Once again the idea is very simple. We view the goal pose as a point of low potential energy and the

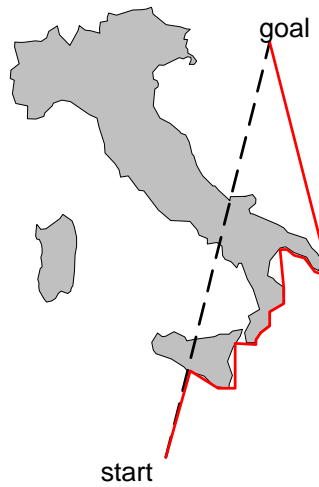



Figure 2.8: The visbug algorithm executing a goal-seek trajectory around Southern Italy.

obstacles as areas of high potential energy. If we think of the vehicle as a ball-bearing free to roll around on a "potential terrain" then it will naturally roll around obstacles and down towards the goal point. The local curvature and magnitude of the potential field can be used to deduce a locally preferred motion. We now firm up this intuitive description with some mathematics.

At any point $\mathbf{x}$ we can write the total potential $\mathbf{U}_\Sigma$ as a sum of the potential induced $\mathbf{U}_o$ by k obstacles and the potential induced by the goal $\mathbf{U}_g$:

$$\mathbf{U}_\Sigma(\mathbf{x}) = \sum_{i=1:k} \mathbf{U}_{o,i}(\mathbf{x}) + \mathbf{U}_g(\mathbf{x}) \tag{2.7}$$

Now we know that the force $\mathbf{F}(\mathbf{x})$ exerted on a particle in a potential field $\mathbf{U}_\Sigma(\mathbf{x})$ can be written as :

$$\mathbf{F}(\mathbf{x}) = -\nabla \mathbf{U}_\Sigma(\mathbf{x}) \tag{2.8}$$

$$= -\sum_{i=1:k} \nabla \mathbf{U}_{o,i}(\mathbf{x}) - \nabla \mathbf{U}_g(\mathbf{x}) \tag{2.9}$$

where $\nabla$is the grad operator $\nabla \mathbf{V} = \mathbf{i}\frac{\partial \mathbf{V}}{\partial x} + \mathbf{j}\frac{\partial \mathbf{V}}{\partial y} + \mathbf{k}\frac{\partial \mathbf{V}}{\partial x}$ [5] This is a powerful tool - we can simply move the vehicle in the manner in which a particle would move in a location-dependent force-field. Equation 2.8 tells us the direction and magnitude of the force for any vehicle position $\mathbf{x}$.

[5] don't get confused here with the ∇ notation used for jacobians in the material covering non-linear estimation in coming pages!

The next question is what exactly do the potential functions look like. Well, there is no single answer to that — you can "roll your own". A good choice would be to make the potential of an obstacle be an inverse square function of the distance between vehicle and obstacle. We may also choose an everywhere-convex potential for the goal so that where ever we start, in the absence of obstacles, we will "fall" towards the goal point. Figure 2.6 shows two useful potential candidates (forgive the pun). Defining $\rho(\mathbf{x})$ as the shortest distance

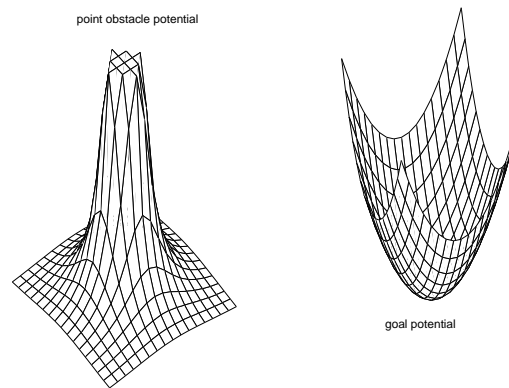



Figure 2.9: Two typical potential functions - inverse quadratic for obstacle and quadratic for the goal.

between the obstacle and the vehicle (at $\mathbf{x}$) and $\mathbf{x}_g$ as the goal point, the algebraic functions for these potentials are:

$$\mathbf{U}_{o,i}(\mathbf{x}) = \eta \begin{cases} \frac{1}{2}\left(\frac{1}{\rho(\mathbf{x})} - \frac{1}{\rho_0}\right)^2 & \forall \quad \rho(\mathbf{x}) \leq \rho_0 \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

$$\mathbf{U}_g(\mathbf{x}) = \frac{1}{2}(\mathbf{x} - \mathbf{x}_g)^2 \tag{2.11}$$

The term ρ_0 places a limit on the region of space affected by the potential field — the virtual vehicle is only affected when it comes with ρ_0 of an obstacle. η is just a scale factor. Simple differentiation allows us to write the force vector exerted on a virtual point vehicle as:

$$\mathbf{F}_{o,i} = \begin{cases} \eta\left(\frac{1}{\rho(\mathbf{x})} - \frac{1}{\rho_0}\right)\frac{1}{\rho(\mathbf{x})^2}\frac{\partial\rho(\mathbf{x})}{\partial\mathbf{x}} & \forall \rho(\mathbf{x}) \leq \rho_0 \\ 0 & \text{otherwise} \end{cases} \tag{2.12}$$

where $\frac{\partial\rho(\mathbf{x})}{\partial\mathbf{x}}$ is the vector of derivatives of the distance function $\rho(\mathbf{x})$ with respect to x, y, z. We proceed similarly for the goal potential. So to figure out the force acting on a virtual vehicle and hence the direction the real vehicle should move in, we take the sum of all obstacle

forces and the goal force – a very simple algorithm. The course web-site has example code for a potential path planner written in Matlab. Download it and play.

Although elegant, the potential field method has one serious drawback — it only acts locally. There is no global path planning at play here — the vehicle simply reacts to local obstacles, always moving in a direction of decreasing potential. This policy can lead to the vehicle getting trapped in a local minimum as shown in figure 2.6. Here the vehicle will descend into a local minima in front of the two features and will stop. Any other motion will increase its potential. If you run the code you will see the vehicle getting stuck between two features from time to time [6]. The potential method is someway between the bug method and Voronoi method in terms of distance to obstacles. The bug algorithm sticks close to them, the Voronoi as far away as possible. The potential method simply directs the vehicle in a straight line to a goal unless it moves into the vicinity of of an obstacle in which case it is deflected.

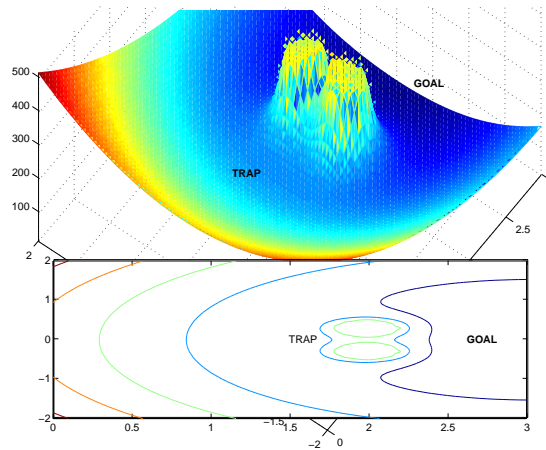



Figure 2.10: A pathological potential terrain and goal combination. The vehicle can get stuck in a local-minima or "trap"

[6] one way out of this is to detect a trap and temporarily move the goal point, the problem is, to where?

Topic 3

Estimation - A Quick Revision

3.1 Introduction

This lecture will begin to cover a topic central to mobile robotics — Estimation. There is a vast literature on Estimation Theory encompassing a rich variation of techniques and ideas. We shall focus on some of the most commonly used techniques which will provide the tools to undertake a surprisingly diverse set of problems. We shall bring them to bear on real life mobile-robot problems. Although we shall discuss and illustrate these techniques in a domain specific fashion, you will do well to keep in your mind that these are general techniques and can be applied to loads of other non-robotic problems[1].

3.2 What is Estimation?

To make sure we are all on the same page it is worth stating (in a wordy fashion) what we mean by "Estimation":

> *"Estimation is the process by which we infer the value of a quantity of interest,* $\mathbf{x}$*, by processing data that is in some way dependent on* $\mathbf{x}$ *."*

[1]For this reason some of this material was previously taught in the B4-Estimation class

There is nothing surprising there —it fits intuitively with our everyday meaning of the word. However, we may have our interest piqued when we impose the following additional characteristics to an estimator.

- We expect the measurements to be noise corrupted and would expect to see this uncertainty in input transformed to uncertainty in inference.
- We do not expect the measurements to always be of $\mathbf{x}$ directly. For example a GPS receiver processes measurements of time (4 or more satellites transmit "time-of-day-at-source") and yet it infers Euclidean position.
- We might like to incorporate prior information in our estimation. For example we may know the depth and altitude of a submarine but not its latitude.
- We might like to employ a model that tells us how we expect a system to evolve over time. For example given a speed and heading we could reasonably offer a model on how a ship might move between two time epochs — open-loop.
- We expect these models to be uncertain. Reasonably we expect this modelling error to be handled in the estimation process and manifest itself in the uncertainty of the estimated state.

Pretty quickly we see that if we can build an estimator capable of handling the above conditions and requirements we are equipping ourselves with a powerful and immensely useful inference mechanism.

Uncertainty is central to the problem of estimation (and robotics). After all, if it weren't for uncertainty many problems would have a simple algebraic solution —"give me the distances to three known points and the algebraically-determined intersection of three circles will define my location". We will be trying to find answers to more realistic questions like "I have three measurements (all with different uncertainty) to three surveyed points (known roughly) — what is your best estimate of my location?". Clearly this is a much harder and more sophisticated question. But equipped with basic probability theory and a little calculus the next sections will derive techniques capable of answering these questions using a few modest assumptions.

3.2.1 Defining the problem

Probability theory and random-variables are the obvious way to mathematically manipulate and model uncertainties and as such much of this lecture will rely on your basic knowledge

of these subjects. We however undertake a quick review.

We want to obtain our best estimate $\hat{\mathbf{x}}$ for a parameter $\mathbf{x}$ given a set of k measurements $\mathbf{Z}^k = \{\mathbf{z}_1, \mathbf{z}_2 \cdots \mathbf{z}_k\}$. We will use a "hat" to denote an estimated quantity and allow the absence of a hat to indicate the true (and unknowable) state of the variable.

We will start by considering four illuminating estimation problems - Maximum Likelihood, Maximum a-posterior, Least Squares and Minimum Mean Squared Error.

3.3 Maximum Likelihood Estimation

It is sane to suppose that a measurement $\mathbf{z}$ that we are given is in some way related to the state we wish to estimate [2]. We also suppose that measurements (also referred to as observations in these notes) are not precise and are noise corrupted. We can encapsulate both the relational and uncertain aspects by **defining** a **likelihood** function:

$$\mathcal{L} \triangleq p(\mathbf{z}|\mathbf{x}) \tag{3.1}$$

The distribution $p(\mathbf{z}|\mathbf{x})$ is the *conditional* probability of the measurement $\mathbf{z}$ given a particular value of x. Figure 3.3 is just such a distribution — a Gaussian in this case(C is just a normalising constant):

$$p(\mathbf{z}|\mathbf{x}) = \frac{1}{C} e^{-\frac{1}{2}(\mathbf{z}-\mathbf{x})^T \mathbf{P}^{-1}(\mathbf{z}-\mathbf{x})} \tag{3.2}$$

Notice that equation 3.2 is a function of both $\mathbf{x}$ and $\mathbf{z}$. Crucially we interpret $\mathcal{L}$ as function of $\mathbf{x}$ and not $\mathbf{z}$ as you might initially think. Imagine we have been given an observation $\mathbf{z}$ and an associated pdf $\mathcal{L}$ for which we have a functional form of (Equation 3.2). We form our maximum likelihood estimate $\hat{\mathbf{x}}_{m.l}$ by varying $\mathbf{x}$ until we find the maximum likelihood.

Given an observation $\mathbf{z}$ and a likelihood function $p(\mathbf{z}|\mathbf{x})$, the **maximum likelihood estimator - ML** finds the value of $\mathbf{x}$ which maximises the likelihood function $\mathcal{L} \triangleq p(\mathbf{z}|\mathbf{x})$.

$$\hat{\mathbf{x}}_{m.l} = \arg\max_{\mathbf{x}} p(\mathbf{z}|\mathbf{x}) \tag{3.3}$$

[2] clearly, as if they were independent $\mathbf{Z^k}$ would contain no information about $\mathbf{x}$ and the sensor providing the measurements wouldn't be much good

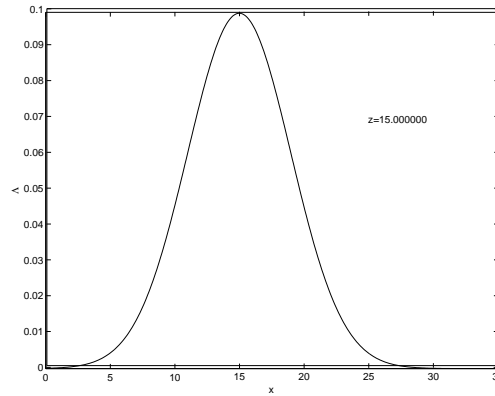



Figure 3.1: A Gaussian Likelihood function for a scalar observation $\mathbf{z}$

3.4 Maximum A-Posteriori - Estimation

In many cases we will already have some prior knowledge on $\mathbf{x}$. Imagine $\mathbf{x}$ is a random variable which we have good reason to suppose is distributed as p($\mathbf{x}$). For example, perhaps we know the intended (by design) rotational velocity μ_p of a CDROM drive - we might model our prior knowledge of this as a gaussian $\sim \mathcal{N}(\mu_p, \sigma_p^2)$. If we have a sensor that can produce an observation $\mathbf{z}$ of actual rotation speed with a behaviour modelled as $p(\mathbf{z}|\mathbf{x})$ we can use Bayes rule to come up with a *posterior* pdf $p(\mathbf{x}|\mathbf{z})$ that incorporates not only the information from the sensor but also our assumed prior knowledge on $\mathbf{x}$:

$$\begin{aligned} p(\mathbf{x}|\mathbf{z}) &= \frac{p(\mathbf{z}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{z})} \\ &= C \times p(\mathbf{z}|\mathbf{x})p(\mathbf{x}) \end{aligned}$$

The **maximum a-posteriori - MAP** finds the value of $\mathbf{x}$ which maximises $p(\mathbf{z}|\mathbf{x})p(\mathbf{x})$ (the normalising constant is not a function of $\mathbf{x}$).

Given an observation $\mathbf{z}$,a likelihood function $p(\mathbf{z}|\mathbf{x})$ and a prior distribution on $\mathbf{x}$, $p(\mathbf{x})$, the **maximum a posteriori estimator - MAP** finds the value of $\mathbf{x}$ which maximises the posterior distribution $p(\mathbf{x}|\mathbf{z})$

$$\hat{\mathbf{x}}_{map} = \arg\max_{\mathbf{x}} p(\mathbf{z}|\mathbf{x})p(\mathbf{x}) \tag{3.4}$$

Lets quickly finish the CD example, assume our sensor also has a gaussian likelihood function, centered on the observation $\mathbf{z}$ but with a variance σ_z^2:

$$\begin{aligned}
p(\mathbf{x}) &= C_1 \exp\{-\frac{(\mathbf{x}-\mu_p)^2}{2\sigma_p^2}\} \\
p(\mathbf{z}|\mathbf{x}) &= C_2 \exp\{-\frac{(\mathbf{z}-\mathbf{x})^2}{2\sigma_z^2}\} \\
p(\mathbf{z}|\mathbf{x}) &= \frac{p(\mathbf{z}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{z})} \\
&= C(\mathbf{z}) \times p(\mathbf{z}|\mathbf{x}) \times p(\mathbf{x}) \\
&= C(\mathbf{z}) \exp\{-\frac{(\mathbf{x}-\mu_p)^2}{2\sigma_p^2} - \frac{(\mathbf{z}-\mathbf{x})^2}{2\sigma_z^2}\}
\end{aligned}$$

$p(\mathbf{z}|\mathbf{x})$ is maximum when the exponent is zero. A simple way to find what value of $\mathbf{x}$ achieves this is to re-write the exponent of $p(\mathbf{z}|\mathbf{x})$ as

$$\frac{(x-\alpha)^2}{\beta^2} = -\frac{(\mathbf{x}-\mu_p)^2}{2\sigma_p^2} - \frac{(\mathbf{z}-\mathbf{x})^2}{2\sigma_z^2} \tag{3.5}$$

Expanding the R.H.S and comparing coefficients swiftly leads to:

$$\begin{aligned}
\alpha &= \frac{\sigma_z^2\mu_p + \sigma_p^2\mathbf{z}}{\sigma_z^2 + \sigma_p^2} \\
\beta^2 &= \frac{\sigma_z^2\sigma_p^2}{\sigma_z^2 + \sigma_p^2}
\end{aligned}$$

Therefore the MAP estimate $\hat{\mathbf{x}}$ is $\frac{\sigma_z^2\mu_p+\sigma_p^2\mathbf{z}}{\sigma_z^2+\sigma_p^2}$ and it has variance $\frac{\sigma_z^2\sigma_p^2}{\sigma_z^2+\sigma_p^2}$. It is interesting and informative to note that as we increase the prior uncertainty in the CD speed by increasing σ_p towards infinity then $\hat{\mathbf{x}}$ tends towards $\mathbf{z}$ —the ML estimate. In other words when we have an uninformative prior the MAP estimate is the same as the ML estimate. This makes sense as in this case the only thing providing information is the sensor (via its likelihood function).

There several other things that you should be sure you appreciate before progressing:

Decreasing posterior variance Calling the posterior variance σ_+^2 (which is β^2 above), we note it is smaller than that of the prior. In the simple example above this can be seen as $\frac{1}{\sigma_+^2} = \frac{1}{\beta^2} = \frac{1}{\sigma_p^2} + \frac{1}{\sigma_z^2}$. This is not surprising as the measurement is *adding* information [3]

[3] Indeed a useful metric of information is closely related to the inverse of variance — note the addition of inverses...

Update to prior In the Gaussian example above we can also write $\hat{\mathbf{x}}_{map}$ as an adjustment to the prior mean:

$$\hat{\mathbf{x}}_{map} = \mu_p + \frac{\sigma_p^2}{\sigma_p^2 + \sigma_z^2} \times (\mathbf{z} - \mu_p) \tag{3.6}$$

clearly if the observation matches the mode of the prior (expected) distribution then $\hat{\mathbf{x}}_{map}$ is unchanged from the prior (but it's uncertainty still decreases). Also note that the correction $\mathbf{z} - \mu_p$ is scaled by the relative uncertainty in both prior and observation - if the observation variance σ_z^2 is huge compared to σ_p^2 (i.e. the sensor is pretty terrible) then irrespective of the magnitude of $\mathbf{z} - \mu_p$, $\hat{\mathbf{x}}_{map}$ is still μ_p and the decrease in uncertainty is small. This too makes sense — if a sensor is very noisy we should not pay too much attention to disparities between expected (the prior) and measured values. This is a concept that we will meet again soon when we discuss Kalman Filtering.

3.5 Minimum Mean Squared Error Estimation

Another key technique for estimating the value of a random variable $\mathbf{x}$ is that of minimum mean squared error estimation. Here we assume we have been furnished with a set of observations $\mathbf{Z^k}$. We define the following cost function which we try to minimise as a function of $\hat{\mathbf{x}}$:

$$\hat{\mathbf{x}}_{mmse} = \arg\min_{\hat{\mathbf{x}}} \mathcal{E}\{(\hat{\mathbf{x}} - \mathbf{x})^T(\hat{\mathbf{x}} - \mathbf{x})|\mathbf{Z^k}\} \tag{3.7}$$

The motivation is clear, we want to find an estimate of $\mathbf{x}$ that given all the measurement minimises the *expected* value of sum of the squared errors between the truth and estimate. Note also that we are trying to minimise a scalar quantity.

Recall from probability theory that

$$\mathcal{E}\{g(x)|y\} = \int_{-\infty}^{\infty} g(\mathbf{x})p(\mathbf{x}|\mathbf{y})dx \tag{3.8}$$

the cost function is then:

$$J(\hat{\mathbf{x}}, \mathbf{x}) = \int_{-\infty}^{\infty} (\hat{\mathbf{x}} - \mathbf{x})^T(\hat{\mathbf{x}} - \mathbf{x})p(\mathbf{x}|\mathbf{Z^k})d\mathbf{x} \tag{3.9}$$

Differentiating the cost function and setting to zero:

$$\frac{\partial J(\hat{\mathbf{x}}, \mathbf{x})}{\partial \hat{\mathbf{x}}} = 2\int_{-\infty}^{\infty} (\hat{\mathbf{x}} - \mathbf{x})p(\mathbf{x}|\mathbf{Z^k})d\mathbf{x} = 0 \tag{3.10}$$

Splitting apart the integral, noting that $\hat{\mathbf{x}}$ is a constant and rearranging:

$$\int_{-\infty}^{\infty} \hat{\mathbf{x}} p(\mathbf{x}|\mathbf{Z^k}) d\mathbf{x} = \int_{-\infty}^{\infty} \mathbf{x} p(\mathbf{x}|\mathbf{Z^k}) d\mathbf{x} \tag{3.11}$$

$$\hat{\mathbf{x}} \int_{-\infty}^{\infty} p(\mathbf{x}|\mathbf{Z^k}) d\mathbf{x} = \int_{-\infty}^{\infty} \mathbf{x} p(\mathbf{x}|\mathbf{Z^k}) d\mathbf{x} \tag{3.12}$$

$$\hat{\mathbf{x}} = \int_{-\infty}^{\infty} \mathbf{x} p(\mathbf{x}|\mathbf{Z^k}) d\mathbf{x} \tag{3.13}$$

$$\hat{\mathbf{x}}_{mmse} = \mathcal{E}\{\mathbf{x}|\mathbf{Z^k}\} \tag{3.14}$$

Why is this important? - it tells us that the mmse estimate of a random variable given a whole bunch of measurements is just the mean of that variable conditioned on the measurements. We shall use this result time and time again in coming derivations.

Why is it different from the Least Squares Estimator? They are related (the LSQ estimator can be derived from Bayes rule) but here $\mathbf{x}$ is a random variable where in the LSQ case $\mathbf{x}$ was a constant unknown. Note we haven't discussed the LSQ estimator yet - but by the time you come to revise from these notes you will have.

3.6 Recursive Bayesian Estimation

The idea of MAP estimation leads naturally to that of recursive Bayesian estimation. In MAP estimation we fused both *a-priori* beliefs and current measurements to come up with an estimate, $\hat{\mathbf{x}}$, of the underlying world state $\mathbf{x}$. If we then took another measurement we could use our previous $\hat{\mathbf{x}}$ as the prior, incorporate the new measurement and come up with a fresh posterior based now, on two observations. The appeal of this approach to a robotics application is obvious. A sensor (laser radar etc) produces a time-stamped sequence of observations. At each time step k we would like to obtain an estimate for it's state *given all* observations up to that time (the set $\mathbf{Z^k}$). We shall now use Bayes rule to frame this more precisely:

$$p(\mathbf{x}, \mathbf{Z^k}) = p(\mathbf{x}|\mathbf{Z^k}) p(\mathbf{Z^k}) \tag{3.15}$$

and also

$$p(\mathbf{x}, \mathbf{Z^k}) = p(\mathbf{Z^k}|\mathbf{x}) p(\mathbf{x}) \tag{3.16}$$

so

$$p(\mathbf{x}|\mathbf{Z^k})p(\mathbf{Z^k}) = p(\mathbf{Z^k}|\mathbf{x})p(\mathbf{x}) \tag{3.17}$$

if we assume (reasonably) that given the underlying state, observations are conditionally independent we can write

$$p(\mathbf{Z^k}|x) = p(\mathbf{Z}^{k-1}|\mathbf{x})p(\mathbf{z}_k|\mathbf{x}) \tag{3.18}$$

where $\mathbf{z}_k$ is a single observation arriving at time k. Substituting into 3.17 we get

$$p(\mathbf{x}|\mathbf{Z^k})p(\mathbf{Z^k}) = p(\mathbf{Z}^{k-1}|\mathbf{x})p(\mathbf{z}_k|\mathbf{x})p(\mathbf{x}) \tag{3.19}$$

now we invoke Bayes rule to re-express the $p(\mathbf{Z}^{k-1}|\mathbf{x})$ term

$$p(\mathbf{Z}^{k-1}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{Z}^{k-1})p(\mathbf{Z}^{k-1})}{p(\mathbf{x})} \tag{3.20}$$

and substituting to reach

$$\begin{aligned} p(\mathbf{x}|\mathbf{Z^k})p(\mathbf{Z^k}) &= p(\mathbf{z}_k|\mathbf{x})\frac{p(\mathbf{x}|\mathbf{Z}^{k-1})p(\mathbf{Z}^{k-1})}{p(\mathbf{x})}p(\mathbf{x}) && (3.21) \\ &= p(\mathbf{z}_k|\mathbf{x})p(\mathbf{x}|\mathbf{Z}^{k-1})p(\mathbf{Z}^{k-1}) && (3.22) \end{aligned}$$

so

$$p(\mathbf{x}|\mathbf{Z^k}) = \frac{p(\mathbf{z}_k|\mathbf{x})p(\mathbf{x}|\mathbf{Z}^{k-1})p(\mathbf{Z}^{k-1})}{p(\mathbf{Z^k})} \tag{3.23}$$

note that

$$\begin{aligned} p(\mathbf{z}_k|\mathbf{Z}^{k-1}) &= \frac{p(\mathbf{z}_k, \mathbf{Z}^{k-1})}{p(\mathbf{Z}^{k-1})} && (3.24) \\ &= \frac{p(\mathbf{Z^k})}{p(\mathbf{Z}^{k-1})} && (3.25) \end{aligned}$$

so

$$\frac{p(\mathbf{Z}^{k-1})}{p(\mathbf{Z^k})} = \frac{1}{p(\mathbf{z}_k|\mathbf{Z}^{k-1})} \tag{3.26}$$

finally then we arrive at our desired result:

$$p(\mathbf{x}|\mathbf{Z^k}) = \frac{p(\mathbf{z}_k|\mathbf{x})p(\mathbf{x}|\mathbf{Z}^{k-1})}{p(\mathbf{z}_k|\mathbf{Z}^{k-1})}$$

where the denominator is just a normaliser

(3.27)

So what does this tell us? Well, we recognise $p(\mathbf{x}|\mathbf{Z^k})$ as our goal - the pdf of $\mathbf{x}$ conditioned on all observations we have received up to and including time k. The $p(\mathbf{z}_k|\mathbf{x})$ term is just the likelihood of the k^{th} measurement. Finally $p(\mathbf{x}|\mathbf{Z}^{k-1})$ is a prior — it is our last best estimate of $\mathbf{x}$ at time $k-1$ which at that time was conditioned on all the $k-1$ measurements that had been made up until that time. The recursive Bayesian estimator is a powerful mechanism allowing new information to be added simply by multiplying a prior by a (current) likelihood.

Note that nothing special has been said about the form of the distributions manipulated in the above derivation. The relationships are true whatever the form of the pdfs. However we can arrive at a very useful result if we consider the case where we assume Gaussian priors and likelihoods... the linear Kalman Filter.

Topic 4

Least Squares Estimation

4.1 Motivation

Imagine we have been given a vector of measurements $\mathbf{z}$ which we believe is related to a state vector of interest $\mathbf{x}$ by the linear equation

$$\mathbf{z} = \mathbf{H}\mathbf{x} \tag{4.1}$$

We wish to take this data and solve this equation to find $\mathbf{x}$ in terms of $\mathbf{z}$. Initially you may naively think that a valid solution is

$$\mathbf{x} = \mathbf{H}^{-1}\mathbf{z} \tag{4.2}$$

which is only a valid solution if $\mathbf{H}$ is a square matrix with $\mid \mathbf{H} \mid \neq 0$ — $\mathbf{H}$ must be invertible. We can get around this problem by seeking a solution $\hat{\mathbf{x}}$ that is closest to the ideal[1] The metric of "closest" we choose is the following:

$$\hat{\mathbf{x}} = \arg\min_{x} \mid\mid \mathbf{H}\mathbf{x} - \mathbf{z} \mid\mid^2 \tag{4.3}$$

$$\hat{\mathbf{x}} = \arg\min_{x} \left\{ (\mathbf{H}\mathbf{x} - \mathbf{z})^T (\mathbf{H}\mathbf{x} - \mathbf{z}) \right\} \tag{4.4}$$

Equation 4.4 can be seen to be a "least squares" criterion. There are several ways to solve this problem we will describe two of them - one appealing to geometry and one to a little calculus.

[1] For this section we will assume that we have more observations than required ie $\dim(\mathbf{z}) > \dim(\mathbf{x})$ which assures that there is a unique "best" solution

4.1.1 A Geometric Solution

Recall from basic linear algebra that the vector $\mathbf{Hx}$ is a linear sum of the columns of $\mathbf{H}$. In other words $\mathbf{Hx}$ ranges over the column space of $\mathbf{H}$. We seek a vector $\mathbf{x}$ such that $\mathbf{Hx}$ is closest to the data vector $\mathbf{z}$. This is achieved when the error vector $\mathbf{e} = \mathbf{Hx} - \mathbf{z}$ is orthogonal to the space in which $\mathbf{Hx}$ is embedded. Thus $\mathbf{e}$ must be orthogonal to every column of $\mathbf{H}$:

$$\mathbf{H}^T(\mathbf{Hx} - \mathbf{z}) = \mathbf{0} \tag{4.5}$$

which can be rearranged as

$$\mathbf{H}^T\mathbf{Hx} = \mathbf{H}^T\mathbf{z} \tag{4.6}$$

$$\mathbf{x} = (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{z} \tag{4.7}$$

This is the least squares solution for $\mathbf{x}$. The matrix $(\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T$ is called the pseudo-inverse of $\mathbf{H}$.

4.1.2 LSQ Via Minimisation

We can expand and set the derivative of Equation 4.4 to zero :

$$\| \mathbf{Hx} - \mathbf{z} \|^2 = \mathbf{x}^T\mathbf{H}^T\mathbf{Hx} - \mathbf{x}^T\mathbf{H}^T\mathbf{z} - \mathbf{zHx} + \mathbf{z}^T\mathbf{z} \tag{4.8}$$

$$\frac{\partial \| \mathbf{Hx} - \mathbf{z} \|^2}{\partial \mathbf{x}} = 2\mathbf{H}^T\mathbf{Hx} - 2\mathbf{H}^T\mathbf{z} \tag{4.9}$$

$$= 0$$

$$\Rightarrow \mathbf{x} = (\mathbf{H}^T\mathbf{H})^{-1}\mathbf{H}^T\mathbf{z} \tag{4.10}$$

The least squares solution for the linear system $\mathbf{Ax} = \mathbf{b}$ with $\mathbf{A}_{m,n} \quad m > n$ is

$$\hat{\mathbf{x}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b} \tag{4.11}$$

4.2 Weighted Least Squares

Imagine now that we have some information regarding how reliable each of the elements in $\mathbf{z}$ is. We might express this information as a diagonal measurement covariance matrix $\mathbf{R}$:

$$\mathbf{R} = \begin{bmatrix} \sigma_{z1}^2 & 0 & 0 \\ 0 & \sigma_{z2}^2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \tag{4.12}$$

It would be natural to weight each element of the error vector $\mathbf{e}$ according to our uncertainty in each element of the measurement vector $\mathbf{z}$ - ie by $\mathbf{R}^{-1}$. The new minimisation becomes:

$$\hat{\mathbf{x}} = \arg\min_x \mid\mid \mathbf{R}^{-1}(\mathbf{H}\mathbf{x} - \mathbf{z}) \mid\mid^2 \tag{4.13}$$

Carrying this through the same analysis yields the **weighted linear least squares estimate**:

$$\hat{\mathbf{x}} = (\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})^{-1}\mathbf{H}\mathbf{R}^{-1}\mathbf{z} \tag{4.14}$$

4.2.1 Non-linear Least Squares

The previous section allows us to derive a least squares estimate under a linear observation model. However most interesting problems will involve non-linear models - measuring the Euclidean distance between two points for example. We now have a new minimisation task:

$$\hat{\mathbf{x}} = \arg\min_x \mid\mid \mathbf{h}(\mathbf{x}) - \mathbf{z} \mid\mid^2 \tag{4.15}$$

We begin by assuming we have some initial guess of a solution $\mathbf{x}_0$. We seek a vector δ such that $\mathbf{x}_1 = \mathbf{x}_0 + \delta$ and $\mid\mid \mathbf{h}(\mathbf{x}_1) - \mathbf{z} \mid\mid^2$ is minimised. To proceed we use a Taylor series expansion:

$$\mathbf{h}(\mathbf{x}_0 + \delta) = \mathbf{h}(\mathbf{x}_0) + \nabla\mathbf{H}_{\mathbf{x_0}}\delta \tag{4.16}$$

$$\mid\mid \mathbf{h}(\mathbf{x}_1) - \mathbf{z} \mid\mid^2 = \mid\mid \mathbf{h}(\mathbf{x}_0) + \nabla\mathbf{H}_{\mathbf{x_0}}\delta - \mathbf{z} \mid\mid^2 \tag{4.17}$$

$$= \mid\mid \underbrace{\nabla\mathbf{H}_{\mathbf{x_0}}}_{\mathbf{A}}\delta - \underbrace{(\mathbf{z} - \mathbf{h}(\mathbf{x}_0))}_{\mathbf{b}} \mid\mid^2 \tag{4.18}$$

where

$$\nabla\mathbf{H}_{\mathbf{x_0}} = \frac{\partial\mathbf{h}}{\partial\mathbf{x}} = \underbrace{\begin{bmatrix} \frac{\partial\mathbf{h}_1}{\partial\mathbf{x}_1} & \cdots & \frac{\partial\mathbf{h}_1}{\partial\mathbf{x}_m} \\ \vdots & & \vdots \\ \frac{\partial\mathbf{h}_n}{\partial\mathbf{x}_1} & \cdots & \frac{\partial\mathbf{h}_n}{\partial\mathbf{x}_m} \end{bmatrix}}_{\text{evaluated at } \mathbf{x}_0} \tag{4.19}$$

Equation 4.18 can be seen to be a linear least square problem - something we have already solved and stated in equation 4.11. By inspection then we can write an expression for δ that minimises the right hand side of 4.18:

$$\delta = \left(\nabla \mathbf{H}_{\mathbf{x_0}}{}^T \nabla \mathbf{H}_{\mathbf{x_0}}\right)^{-1} \nabla \mathbf{H}_{\mathbf{x_0}}{}^T [\mathbf{z} - \mathbf{h}(\mathbf{x}_0)] \tag{4.20}$$

We now set $\mathbf{x}_1 = \mathbf{x}_0 + \delta$ and iterate again until the norm falls below a tolerance value. Like the linear case, there is a natural extension to the weighted non linear case. For a measurement $\mathbf{z}$ with variance $\mathbf{R}$ the weighted non-linear least squares algorithm is as follows:

1. Begin with an initial guess $\hat{\mathbf{x}}$
2. Evaluate
$$\delta = \left(\nabla \mathbf{H}_{\hat{\mathbf{x}}}{}^T \mathbf{R}^{-1} \nabla \mathbf{H}_{\hat{\mathbf{x}}}\right)^{-1} \nabla \mathbf{H}_{\hat{\mathbf{x}}}{}^T \mathbf{R}^{-1} [\mathbf{z} - \mathbf{h}(\hat{\mathbf{x}})]$$
3. Set $\hat{\mathbf{x}} = \hat{\mathbf{x}} + \delta$
4. If $\| \mathbf{h}(\hat{\mathbf{x}}) - \mathbf{z} \|^2 > \epsilon$ goto 1 else stop.

4.2.2 Long Baseline Navigation - an Example

So why is the non-linear LSQ problem interesting? Well, non-linear least squares allows us to solve some interesting and realistic navigation problems. For example, consider the case of an autonomous underwater vehicle (AUV) moving within a network of acoustic beacons. The AUV shown in figure 4.2.2 is about to be launched on a mine detection mission in the Mediterranean. Within the hull (which floods) is a transceiver which emits a "call" pulse into the water column. Beacons deployed at known locations detect this pulse and reply with "acknowledge" pulses which are detected by the transceiver. The difference in time between "call" and "acknowledge" is proportional to the distance between vehicle and each beacon. Figure 4.2.2 shows a diagram of this kind of navigation (which is very similar to how GPS works).

Imagine the vehicle is operating within a network of 4 beacons. We wish to find an LSQ estimate of the vehicle's position $\mathbf{x}_v = [x, y, z]^T$. Each beacon i is at known position $\mathbf{x}_{bi} = [x_{bi}, y_{bi}, z_{bi}]^T$. We shall assume that the observations become available simultaneously

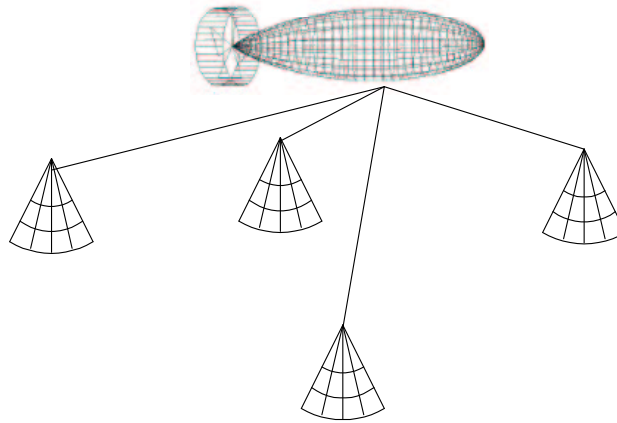



Figure 4.1: A Long Baseline Acoustic Network for an Autonomous Underwater Vehicle (AUV)

so we can stack them into a single vector[2]. The model of the long-baseline transceiver operating in water with speed of sound c can be written as follows:

$$\mathbf{z} = \begin{bmatrix} t_1 & t_2 & t_3 & t_4 \end{bmatrix}^T = h(\mathbf{x}_v) \tag{4.21}$$

$$\begin{bmatrix} t_1 \\ t_2 \\ t_3 \end{bmatrix} = \frac{2}{c} \begin{bmatrix} || \mathbf{x}_{b1} - \mathbf{x}_v || \\ || \mathbf{x}_{b2} - \mathbf{x}_v || \\ || \mathbf{x}_{b3} - \mathbf{x}_v || \\ || \mathbf{x}_{b4} - \mathbf{x}_v || \end{bmatrix} \tag{4.22}$$

so

$$\nabla \mathbf{H}_{\mathbf{x_v}} = -\frac{2}{rc} \begin{bmatrix} \Delta_{x1} & \Delta_{y1} & \Delta_{y1} \\ \Delta_{x2} & \Delta_{y2} & \Delta_{y2} \\ \Delta_{x3} & \Delta_{y3} & \Delta_{y3} \\ \Delta_{x4} & \Delta_{y4} & \Delta_{y4} \end{bmatrix} \tag{4.23}$$

where

$$\begin{aligned} \Delta_{xi} &= x_{bi} - x \\ \Delta_{yi} &= y_{bi} - y \\ \Delta_{zi} &= z_{bi} - z \\ r &= \sqrt{(x_{bi} - x)^2) + (y_{bi} - y)^2) + (z_{bi} - z)^2)} \end{aligned}$$

[2]In practice of course the signal that travels the furthest comes in last - the measurements are staggered. This is only a problem if the vehicle is moving

Figure 4.2: An autonomous underwater vehicle about to be launched. The large fork on the front is a sonar designed to detect buried mines. Within the hull is a transceiver which emits a "call" pulse into the water column. Beacons deployed at known locations detect this pulse and reply with a "acknowledge" pulses which are detected by the transceiver. The difference in time between "call" and "acknowledge" is proportional to the distance between vehicle and each beacon. (photo courtesy of MIT) see figure 4.2.2

With the jacobian calculated and given a set of four measurements to the beacons we can iteratively apply the non-linear least squares algorithm until the change in $\mathbf{x}_v$ between iterations becomes small enough to disregard. You might be surprised that many of the oil rigs floating in the Gulf of Mexico basically use this method to calculate their position relative to the well-head thousands of meters below them. Oil rigs are perhaps one of the largest mobile-robots you are likely to encounter.

One issue with the Least Squares solution is that enough data has to be accumulated to make the system observable - $\mathbf{H}^T\mathbf{H}$ must be invertible. For example, in our subsea example, acoustic-refractive properties of the water column may mean that replies from a particular beacon are never detected. Alternatively acoustic noise may obliterate the true signal leading to false detections. Figure 4.2.2 shows some real LBL data from vehicle shown in Figure 4.2.2. Most of the noise is due to the enormously powerful fork-like sonar on its nose.

The Kalman filter which we shall discuss, derive and use shortly is one way to overcome this problem of instantaneous un-observability.

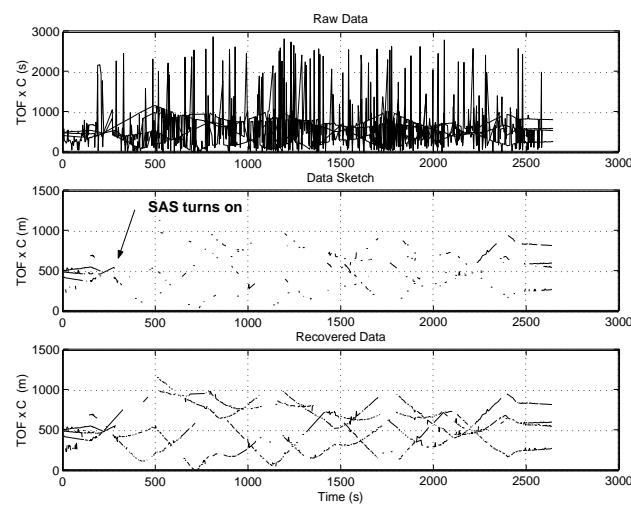



Figure 4.3: Some real data from the ocean. The synthetic aperture adds a lot of noise to the LBL sensing scheme. However it is possible to separate true signal from noise as shown in the lower graphs. The y axes are time of flight $\times$ c giving effective distance between vehicle and beacons.

Topic 5

Kalman Filtering -Theory, Motivation and Application

Keep in mind

The following algebra is simple but a little on the laborious side and was provided here for completeness. The following section will involve a linear progression of simple algebraic steps. I hope you will derive some degree of intellectual satisfaction in seeing the equations being derived using only Bayes rule. Of course, like many things in engineering, derivations need not be done at every invocation of a smart idea but it is important to understand the underlying principles. The exams will not ask for a complete derivation of the Kalman filter but may ask you to explain the underlying concepts with reference to the key equations.

5.1 The Linear Kalman Filter

We will now assume that the likelihood $p(\mathbf{z}|\mathbf{x})$ and a prior $p(\mathbf{x})$ on $\mathbf{x}$ are Gaussian. Furthermore, initially, we are going to model our sensor as something that produces a observation $\mathbf{z}$ which is a noise corrupted linear function of the state $\mathbf{x}$:

$$\mathbf{z} = \mathbf{H}\mathbf{x} + \mathbf{w} \tag{5.1}$$

Here $\mathbf{w}$ is a gaussian noise with zero mean and covariance $\mathbf{R}$ so that

$$p(\mathbf{w}) = \frac{1}{(2\pi)^{n/2} \mid \mathbf{R} \mid^{1/2}} \exp\{-\frac{1}{2}\mathbf{w}^T\mathbf{R}^{-1}\mathbf{w}\}. \tag{5.2}$$

Note that up until now the examples have often been dealing with scalar 1D distributions. Here we generalise to the multidimensional case but things should still look familiar to you. We shall let the state have n_x dimensions and the observation vector have n_z dimensions.

This allows us to write down a likelihood function:

$$p(\mathbf{z}|\mathbf{x}) = \frac{1}{(2\pi)^{nz/2} \mid \mathbf{R} \mid^{1/2}} \exp\{-\frac{1}{2}(\mathbf{z} - \mathbf{H}\mathbf{x})^T \mathbf{R}^{-1}(\mathbf{z} - \mathbf{H}\mathbf{x})\} \tag{5.3}$$

.

We can also use a multidimensional Gaussian with mean $\mathbf{x}_\ominus$ and covariance $\mathbf{P}_\ominus$ to describe our prior belief in $\mathbf{x}$:

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{nx/2} \mid \mathbf{P}_\ominus \mid^{1/2}} \exp\{-\frac{1}{2}(\mathbf{x} - \mathbf{x}_\ominus)^T \mathbf{P}_\ominus^{-1}(\mathbf{x} - \mathbf{x}_\ominus)\} \tag{5.4}$$

.

Now we can use Bayes rule to figure out an expression for the posterior $p(\mathbf{x}|\mathbf{z})$.:

$$p(\mathbf{x}|\mathbf{z}) = \frac{p(\mathbf{z}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{z})} \tag{5.5}$$

$$= \frac{p(\mathbf{z}|\mathbf{x})p(\mathbf{x})}{\int_{-\infty}^{\infty} p(\mathbf{z}|\mathbf{x})p(\mathbf{x})d\mathbf{x}} \tag{5.6}$$

$$= \frac{\frac{1}{(2\pi)^{nz/2}|\mathbf{R}|^{1/2}} \exp\{-\frac{1}{2}(\mathbf{z} - \mathbf{H}\mathbf{x})^T \mathbf{R}^{-1}(\mathbf{z} - \mathbf{H}\mathbf{x})\} \frac{1}{(2\pi)^{nx/2}|\mathbf{P}_\ominus|^{1/2}} \exp\{-\frac{1}{2}(\mathbf{x} - \mathbf{x}_\ominus)^T \mathbf{P}_\ominus^{-1}(\mathbf{x} - \mathbf{x}_\ominus)\}}{\mathbf{C}(\mathbf{z})} \tag{5.7}$$

This looks pretty formidable but we do know that we will end up with a gaussian (gaussian × gaussian = gaussian) and scale factors are not important therefore we can disregard the denominator and focus on the product:

$$\exp\{-\frac{1}{2}(\mathbf{z} - \mathbf{H}\mathbf{x})^T \mathbf{R}^{-1}(\mathbf{z} - \mathbf{H}\mathbf{x})\} \exp\{-\frac{1}{2}(\mathbf{x} - \mathbf{x}_\ominus)^T \mathbf{P}_\ominus^{-1}(\mathbf{x} - \mathbf{x}_\ominus)\} \tag{5.8}$$

or equivalently:

$$\exp\{-1/2\left((\mathbf{z} - \mathbf{H}\mathbf{x})^T \mathbf{R}^{-1}(\mathbf{z} - \mathbf{H}\mathbf{x}) + (\mathbf{x} - \mathbf{x}_\ominus)\mathbf{P}_\ominus^{-1}(\mathbf{x} - \mathbf{x}_\ominus)^T\right)\} \tag{5.9}$$

We know (because gaussian × gaussian = gaussian) that we can find a way to express the above exponent in a quadratic way:

$$(\mathbf{x} - \mathbf{x}_\oplus)^T \mathbf{P}_\oplus^{-1}(\mathbf{x} - \mathbf{x}_\oplus) \tag{5.10}$$

.

We can figure out the new mean $\mathbf{x}_\oplus$ and covariance $\mathbf{P}_\oplus$ by expanding expression 5.9 and comparing terms with expression 5.10. Remember we want to find the new mean because Equation 3.14 tells us this will be the MMSE estimate. So expanding 5.9 we have:

$$\mathbf{x}^T\mathbf{P}_\ominus^{-1}\mathbf{x}-\mathbf{x}_\ominus^T\mathbf{P}_\ominus^{-1}\mathbf{x}_\ominus-\mathbf{x}^T\mathbf{P}_\ominus^{-1}\mathbf{x}_\ominus+\mathbf{x}_\ominus^T\mathbf{P}_\ominus^{-1}\mathbf{x}_\ominus+\mathbf{x}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\mathbf{x}-\mathbf{x}^T\mathbf{H}^T\mathbf{R}^{-1}\mathbf{z}-\mathbf{z}\mathbf{R}^{-1}\mathbf{H}\mathbf{x}+\mathbf{z}^T\mathbf{R}^{-1}\mathbf{z} \tag{5.11}$$

Now collecting terms this becomes:

$$\mathbf{x}^T(\mathbf{P}_\ominus^{-1}+\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})\mathbf{x}-\mathbf{x}^T(\mathbf{P}_\ominus^{-1}\mathbf{x}_\ominus+\mathbf{H}^T\mathbf{R}^{-1}\mathbf{z})-(\mathbf{x}^T\mathbf{P}_\ominus^{-1}+\mathbf{z}\mathbf{R}^{-1}\mathbf{H})\mathbf{x}+(\mathbf{x}_\ominus^T\mathbf{P}_\ominus^{-1}\mathbf{x}_\ominus+\mathbf{z}^T\mathbf{R}^{-1}\mathbf{z}) \tag{5.12}$$

Expanding 5.10:

$$\mathbf{x}^T\mathbf{P}_\oplus^{-1}\mathbf{x}-\mathbf{x}^T\mathbf{P}_\oplus^{-1}\mathbf{x}_\oplus-\mathbf{x}_\oplus^T\mathbf{P}_\oplus^{-1}\mathbf{x}+\mathbf{x}_\oplus^T\mathbf{P}_\oplus^{-1}\mathbf{x}_\oplus. \tag{5.13}$$

Comparing first terms in 5.12 and 5.13 we immediately see that

$$\mathbf{P}_\oplus=(\mathbf{P}_\ominus^{-1}+\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})^{-1}. \tag{5.14}$$

Comparing the second terms we see that:

$$\mathbf{P}_\oplus^{-1}\mathbf{x}_\oplus=\mathbf{P}_\ominus^{-1}\mathbf{x}_\ominus+\mathbf{H}^T\mathbf{R}^{-1}\mathbf{z}. \tag{5.15}$$

Therefore we can write the MMSE estimate, $\mathbf{x}_\oplus$ as

$$\mathbf{x}_\oplus=(\mathbf{P}_\ominus^{-1}+\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})^{-1}(\mathbf{P}_\ominus^{-1}\mathbf{x}_\ominus+\mathbf{H}^T\mathbf{R}^{-1}\mathbf{z}). \tag{5.16}$$

We can combine this result with our understanding of the recursive Bayesian filter we covered in section 3.6. Every time a new measurement becomes available we update our estimate and its covariance using the above two equations.

There is something about the above two equations 5.14 and 5.16 that may make them inconvenient — we have to keep inverting our prior covariance matrix which may be computationally expensive if the state-space is large [1]. Fortunately we can do some algebra to come up with equivalent equations that do not involve an inverse.

We begin by stating a block matrix identity. Given matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ the following is true (for non-singular $\mathbf{A}$):

$$(\mathbf{A}+\mathbf{B}\mathbf{C}\mathbf{B}^T)^{-1}=\mathbf{A}^{-1}-\mathbf{A}^{-1}\mathbf{B}(\mathbf{C}^{-1}+\mathbf{B}^T\mathbf{A}^{-1}\mathbf{B})^{-1}\mathbf{B}^T\mathbf{A}^{-1} \tag{5.17}$$

[1] in some navigation applications the dimension of $\mathbf{x}$ can approach the high hundreds

We can immediately apply this to 5.14 to get:

$$\mathbf{P}_\oplus = \mathbf{P}_\ominus - \mathbf{P}_\ominus\mathbf{H}^T(\mathbf{R} + \mathbf{H}\mathbf{P}_\ominus\mathbf{H}^T)^{-1}\mathbf{H}\mathbf{P}_\ominus \tag{5.18}$$
$$= \mathbf{P}_\ominus - \mathbf{W}\mathbf{S}\mathbf{W}^T \tag{5.19}$$
$$\tag{5.20}$$

or

$$= (\mathbf{I} - \mathbf{W}\mathbf{H})\mathbf{P}_\ominus \tag{5.21}$$

where

$$\mathbf{S} = \mathbf{H}\mathbf{P}_\ominus\mathbf{H}^T + \mathbf{R} \tag{5.22}$$
$$\mathbf{W} = \mathbf{P}_\ominus\mathbf{H}^T\mathbf{S}^{-1} \tag{5.23}$$

Now look at the form of the update equation 5.16 it is a linear combination of $\mathbf{x}$ and $\mathbf{z}$. Combining 5.20 with 5.16 we have:

$$\mathbf{x}_\oplus = (\mathbf{P}_\ominus^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})^{-1}(\mathbf{P}_\ominus^{-1}\mathbf{x}_\ominus + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{z}) \tag{5.24}$$
$$= (\mathbf{P}_\ominus^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})^{-1}(\mathbf{H}^T\mathbf{R}^{-1}\mathbf{z}) + (\mathbf{I} - \mathbf{W}\mathbf{H})\mathbf{P}_\ominus(\mathbf{P}_\ominus^{-1}\mathbf{x}_\ominus) \tag{5.25}$$
$$= (\mathbf{P}_\ominus^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})^{-1}(\mathbf{H}^T\mathbf{R}^{-1}\mathbf{z}) + \mathbf{x}_\ominus + \mathbf{W}(-\mathbf{H}\mathbf{x}_\ominus) \tag{5.26}$$
$$= \mathbf{C}\mathbf{z} + \mathbf{x}_\ominus + \mathbf{W}(-\mathbf{H}\mathbf{x}_\ominus) \tag{5.27}$$

where

$$\mathbf{C} = (\mathbf{P}_\ominus^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})^{-1}\mathbf{H}^T\mathbf{R}^{-1} \tag{5.28}$$

Taking a step aside we note that both

$$\mathbf{H}^T\mathbf{R}^{-1}(\mathbf{H}\mathbf{P}_\ominus\mathbf{H}^T + \mathbf{R}) = \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\mathbf{P}_\ominus\mathbf{H}^T + \mathbf{H}^T \tag{5.29}$$

and also

$$(\mathbf{P}_\ominus^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})\mathbf{P}_\ominus\mathbf{H}^T = \mathbf{H}^T + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}\mathbf{P}_\ominus\mathbf{H}^T \tag{5.30}$$

so

$$(\mathbf{P}_\ominus^{-1} + \mathbf{H}^T\mathbf{R}^{-1}\mathbf{H})^{-1}\mathbf{H}^T\mathbf{R}^{-1} = \mathbf{P}_\ominus\mathbf{H}^T(\mathbf{H}\mathbf{P}_\ominus\mathbf{H}^T + \mathbf{R})^{-1} \tag{5.31}$$

therefore

$$\mathbf{C} = \mathbf{P}_\ominus\mathbf{H}^T\mathbf{S}^{-1} \tag{5.32}$$
$$= \mathbf{W}\text{from } 5.23 \tag{5.33}$$

Finally then we have

$$\mathbf{x}_{\oplus} = \mathbf{x}_{\ominus} + \mathbf{W}(\mathbf{z} - \mathbf{H}\mathbf{x}_{\ominus}) \tag{5.34}$$

We can now summarise the update equations for the Linear Kalman Filter:

The update equations are as follows. Given an observation $\mathbf{z}$ with uncertainty (covariance) $\mathbf{R}$ and a prior estimate $\mathbf{x}_{\ominus}$ with covariance $\mathbf{P}_{\ominus}$ the new estimate and covariance are calculated as:

$$\mathbf{x}_{\oplus} = \mathbf{x}_{\ominus} + \mathbf{W}\nu$$
$$\mathbf{P}_{\oplus} = \mathbf{P}_{\ominus} - \mathbf{W}\mathbf{S}\mathbf{W}^T$$

where the **"Innovation"** ν is

$$\nu = \mathbf{z} - \mathbf{H}\mathbf{x}_{\ominus}$$

the **"Innovation Covariance"** $\mathbf{S}$ is given by

$$\mathbf{S} = \mathbf{H}\mathbf{P}_{\ominus}\mathbf{H}^T + \mathbf{R}$$

and the **"Kalman Gain"** $\mathbf{W}$ is given by

$$\mathbf{W} = \mathbf{P}_{\ominus}\mathbf{H}^T\mathbf{S}^{-1}$$

5.1.1 Incorporating Plant Models - Prediction

The previous section showed how an observation vector ($\mathbf{z}$) related in some linear way to a state we wish to estimate ($\mathbf{x}$) can be used to update in a optimal way a prior belief /estimate. This analysis lead us to the so called Kalman update equations. However there is another case which we would like to analyse: given a prior at time $k-1$, a (imperfect) model of a system that models the transition of state from time $k-1$ to k, what is the new estimate at time k?

Mathematically we are being told that the true state behaves in the following fashion:

$$\mathbf{x}(k) = \mathbf{F}\mathbf{x}(k-1) + \mathbf{B}\mathbf{u}(k) + \mathbf{G}\mathbf{v}(k) \tag{5.35}$$

Here we are modelling our uncertainty in our model by adding a linear transformation of a "white noise" [2] term $\mathbf{v}$ with strength (covariance) $\mathbf{Q}$. The term $\mathbf{u}(k)$ is the **control vector** at time k. It models actions taken on the plant that are independent of the state vector itself. For example, it may model an acceleration on a model that assumes constant velocity. (A good example of $\mathbf{u}$ is the steer angle on a car input into the system by a driver (machine or human))

The $i|j$ notation

Our general approach will be based on an inductive argument. However first we shall introduce a little more notation. We have already shown (and used) the fact that given a set $\mathbf{Z^k}$ of k observations the MMSE estimate of $\mathbf{x}$ is

$$\hat{\mathbf{x}}_{mmse} = \mathcal{E}\{\mathbf{x}|\mathbf{Z^k}\} \tag{5.36}$$

We now drop the "mmse" subscript and start to use two parenthesised time indexes i and j. We use $\hat{\mathbf{x}}(i|j)$ to mean the "MMSE estimate of $\mathbf{x}$ at time i given observations up until time j". Incorporating this into Equation 5.36 we have

$$\hat{\mathbf{x}}(i|j) = \mathcal{E}\{\mathbf{x}(i)|\mathbf{Z}^j\}. \tag{5.37}$$

We also use this notation to define a covariance matrix $\mathbf{P}(i|j)$ as

$$\mathbf{P}(i|j) = \mathcal{E}\{(\mathbf{x}(i) - \hat{\mathbf{x}}(i|j))(\mathbf{x}(i) - \hat{\mathbf{x}}(i|j))^T|\mathbf{Z}^j\} \tag{5.38}$$

which is the mean square error of the estimate $\hat{\mathbf{x}}(i|j)$[3].

[2] a random variable distributed according to a zero-mean gaussian pdf

[3] If this sounds confusing focus on the fact that we are maintaining a probability distribution for $\mathbf{x}$. Our estimate at time i using measurements up until time j ($\hat{\mathbf{x}}(i|j)$) is simply the mean of this distribution and it has variance $\mathbf{P}(i|j)$. If our distribution is in fact a Gaussian then these are the only statistics we need to fully characterize the pdf. This is exactly what a Kalman filter does — it maintains the statistics of a pdf that "best" represents $\mathbf{x}$ given a set of measurements and control inputs

$\hat{\mathbf{x}}$ (i—j) is the estimate of $\mathbf{x}$ at time i given measurements up until time j. Commonly you will see the following combinations:

- $\hat{\mathbf{x}}(k|k)$ estimate at time k given all available measurements. Often simply called the **estimate**
- $\hat{\mathbf{x}}(k|k-1)$ estimate at time k given first $k-1$ measurements. This is often called the **prediction**

Now back to our question about incorporating a plant model $\mathbf{F}$ and a control signal $\mathbf{u}$. Imagine at time k we have been provided with MMSE estimate of $\mathbf{x}$ (k-1). Using our new notation we write this estimate and its covariance as $\hat{\mathbf{x}}(k-1|k-1)$ and $\mathbf{P}(k-1|k-1)$. We are also provided with a control signal $\mathbf{u}(k)$ we want to know how to figure out a new estimate $\hat{\mathbf{x}}(k|k-1)$ at time k. Note that we are still only using measurements up until time $(k-1)$ ($i > j$ in Equation 5.37) however we are talking about an estimate at time k. By convention we call this kind of estimate a **"prediction"**. You can think of predicting as an "open loop" step as no measurements of state are used to correct the calculation.

We can use our "conditional expectation result" to progress:

$$\begin{aligned}
\hat{\mathbf{x}}(k|k-1) &= \mathcal{E}\{\mathbf{x}(k|\mathbf{Z}^{k-1}\} && (5.39)\\
&= \mathcal{E}\{\mathbf{F}\mathbf{x}(k-1) + \mathbf{B}\mathbf{u}(k) + \mathbf{G}\mathbf{v}(k)|\mathbf{Z}^{k-1}\} && (5.40)\\
&= \mathbf{F}\mathcal{E}\{\mathbf{x}(k-1)|\mathbf{Z}^{k-1}\} + \mathbf{B}\mathbf{u}(k) + \mathbf{G}\mathcal{E}\{\mathbf{v}(k)|\mathbf{Z}^{k-1}\} && (5.41)\\
&= \mathbf{F}\hat{\mathbf{x}}(k-1|k-1) + \mathbf{B}\mathbf{u}(k) + \mathbf{0} && (5.42)
\end{aligned}$$

This is a both simple and intuitive result. It simply says that the best estimate at time k given measurements up until time $k-1$ is simply the projection of the last best estimate $\hat{\mathbf{x}}(k-1|k-1)$ through the plant model. Now we turn our attention to the propagation of the covariance matrix through the prediction step.

$$\mathbf{P}(k|k-1) = \mathcal{E}\{(\mathbf{x}(k) - \hat{\mathbf{x}}(k|k-1))(\mathbf{x}(k) - \hat{\mathbf{x}}(k|k-1))^T|\mathbf{Z}^{k-1}\} \qquad (5.43)$$

perfoming the subtraction first

$$\begin{aligned}
\mathbf{x}(k) - \hat{\mathbf{x}}(k|k-1) &= (\mathbf{F}\mathbf{x}(k-1) + \mathbf{B}\mathbf{u}(k) + \mathbf{G}\mathbf{v}(k)) - (\mathbf{F}\hat{\mathbf{x}}(k-1|k-1) + \mathbf{B}\mathbf{u}(k)) && (5.44)\\
&= \mathbf{F}(\mathbf{x}(k-1) - \hat{\mathbf{x}}(k-1|k-1)) + \mathbf{G}\mathbf{v}(k) && (5.45)
\end{aligned}$$

so

$$\mathbf{P}(k|k-1) = \mathcal{E}\{(\mathbf{F}(\mathbf{x}(k-1) - \hat{\mathbf{x}}(k-1|k-1)) + \mathbf{G}\mathbf{v}(k))\times \quad (5.46)$$
$$(\mathbf{F}(\mathbf{x}(k-1) - \hat{\mathbf{x}}(k-1|k-1)) + \mathbf{G}\mathbf{v}(k))^T|\mathbf{Z}^{k-1}\} \quad (5.47)$$

Now we are going to make a moderate assumption: that the previous estimate $\hat{\mathbf{x}}(k-1|k-1)$ and the noise vector (modelling inaccuracies in either model or control) are uncorrelated (i.e. $\mathcal{E}\{(\mathbf{x}(k-1) - \hat{\mathbf{x}}(k-1|k-1))\mathbf{v}(k)^T\}$ is the zero matrix). This means that when we expand Equation 5.47 all cross terms between $\hat{\mathbf{x}}(k-1|k-1)$ and $\mathbf{v}(k)$ disappear. So:

$$\mathbf{P}(k|k-1) = \mathbf{F}\mathcal{E}\{(\mathbf{x}(k-1) - \hat{\mathbf{x}}(k-1|k-1))(\mathbf{x}(k-1) - \hat{\mathbf{x}}(k-1|k-1))^T|\mathbf{Z}^{k-1}\}\mathbf{F}^T + \quad (5.48)$$
$$\mathbf{G}\mathcal{E}\{\mathbf{v}(k)\mathbf{v}(k)^T|\mathbf{Z}^{k-1}\}\mathbf{G}^T \quad (5.49)$$
$$\mathbf{P}(k|k-1) = \mathbf{F}\mathbf{P}(k-1|k-1)\mathbf{F}^T + \mathbf{G}\mathbf{Q}\mathbf{G}^T \quad (5.50)$$

This result should also seem familiar to you (remember that if $x \sim N(\mu, \Sigma)$ and $y = Mx$ then $y \sim N(M\mu, M\Sigma M^T)$?).

5.1.2 Joining Prediction to Updates

We are now almost in a position to put all the pieces together. To start with we insert our now temporal-conditional notation into the Kalman update equations. We use $\hat{\mathbf{x}}(k|k-1)$ as the prior $\mathbf{x}_\ominus$ and process an observation $\mathbf{z}$ at time k. The posterior $\mathbf{x}_\oplus$ becomes $\hat{\mathbf{x}}(k|k)$

$$\hat{\mathbf{x}}(k|k) = \hat{\mathbf{x}}(k|k-1) + \mathbf{W}(k)\nu(k)$$
$$\mathbf{P}(k|k) = \mathbf{P}(k|k-1) - \mathbf{W}(k)\mathbf{S}\mathbf{W}(k)^T$$
$$\nu(k) = \mathbf{z}(k) - \mathbf{H}\mathbf{x}(k|k-1)$$
$$\mathbf{S} = \mathbf{H}\mathbf{P}(k|k-1)\mathbf{H}^T + \mathbf{R}$$
$$\mathbf{W}(k) = \mathbf{P}(k|k-1)\mathbf{H}^T\mathbf{S}^{-1}$$

So now we are in a position to write down the "standard Linear Kalman Filter Equations". If the previous pages of maths have started to haze your concentration wake up now as you will need to know and appreciate the following:

Linear Kalman Filter Equations

prediction:

$$\hat{\mathbf{x}}(k|k-1) = \mathbf{F}\hat{\mathbf{x}}(k-1|k-1) + \mathbf{B}\mathbf{u}(k) \tag{5.51}$$

$$\mathbf{P}(k|k-1) = \mathbf{F}\mathbf{P}(k-1|k-1)\mathbf{F}^T + \mathbf{G}\mathbf{Q}\mathbf{G}^T \tag{5.52}$$

update:

$$\hat{\mathbf{x}}(k|k) = \hat{\mathbf{x}}(k|k-1) + \mathbf{W}(k)\nu(k) \tag{5.53}$$

$$\mathbf{P}(k|k) = \mathbf{P}(k|k-1) - \mathbf{W}(k)\mathbf{S}\mathbf{W}(k)^T \tag{5.54}$$

where

$$\nu(k) = \mathbf{z}(k) - \mathbf{H}\hat{\mathbf{x}}(k|k-1) \tag{5.55}$$

$$\mathbf{S} = \mathbf{H}\mathbf{P}(k|k-1)\mathbf{H}^T + \mathbf{R} \tag{5.56}$$

$$\mathbf{W}(k) = \mathbf{P}(k|k-1)\mathbf{H}^T\mathbf{S}^{-1} \tag{5.57}$$

5.1.3 Discussion

There are several characteristics of the Kalman Filter which you should be familiar with and understand well. A firm grasp of these will make your task of implementing a KF for a robotics problem much easier.

Recursion The Kalman Filter is recursive. The output of one iteration becomes the input to the next.

Initialising Initially you will have to provide $\mathbf{P}(0|0)$ and $\hat{\mathbf{x}}(0|0)$. These are initial guesses (hopefully derived with some good judgement)

Structure The Kalman filter has a predictor-corrector architecture. The prediction step is corrected by fusion of a measurement. Note that the innovation ν is a difference between the actual observation and the *predicted* observation ($\mathbf{H}\hat{\mathbf{x}}(k|k-1)$). If these

two are identical then there is no change to the prediction in the update step —both observation and prediction are in perfect agreement. Fusion happens in data space.

Asynchronisity The update step need not happen at every iteration of the filter. If at a given time step no observations are available then the best estimate at time k is simply the prediction $\hat{\mathbf{x}}(k|k-1)$.

Prediction Covariance Inflation Each predict step inflates the covariance matrix — you can see this by the addition of $\mathbf{GQG}^T$. This makes sense sice we are only using a pre-conceived idea of how $\mathbf{x}$ will evolve. By our own admission — the presence of a noise vector in the model — the model is inaccurate and so we would expect certainty to dilate (uncertainty increase) with each prediction.

Update Covariance Deflation Each update step generally[4] deflates the covariance matrix. The subtraction of $\mathbf{WSW}^T$ [5] during each update illustrates this. This too makes sense. Each observation fused contains some information and this is "added" to the state estimate at each update. A metric of information is related to the inverse of covariance — note how in equation 5.16 $\mathbf{H}^T\mathbf{R}^{-1}\mathbf{H}$ is added to the inverse of $\mathbf{P}$, this might suggest to you that information is additive. Indeed, equation 5.16 is the so called "information form" of the Kalman Filter.

Observability The measurement $\mathbf{z}$ need not fully determine the state $\mathbf{x}$ (ie in general $\mathbf{H}$ is not invertible). This is crucial and useful. In a least squares problem at every update there have to be enough measurements to solve for the state $\mathbf{x}$, However the Kalman filter can perform updates with only partial measurements. However to get useful results over time the system needs to be observable otherwise the uncertainty in unobserved state components will grow without bound. Two factors make this possible : the fact that the priors presumably contain some information about the unobserved states (they were observed in some previous epoch) and the role of correlations.

Correlations The covariance matrix $\mathbf{P}$ is highly informative. The diagonals are the principal uncertainties in each of the state vector elements. The off diagonals tell us about the relationships *between* elements of our estimated vector $\hat{\mathbf{x}}$ — how they are correlated. The Kalman filter implicitly uses these correlations to update states that are not observed directly by the measurement model! Let's take a real life example. Imagine we have a model $[\mathbf{F}, \mathbf{B}\,\mathbf{Q}]$ of a plane flying. The model will explain how the plane moves between epochs as a function of both state and control. For example at 100m/s, nose down 10 deg, after 100ms the plane will have travelled 10m forward (y direction) and perhaps 1.5 m down (in z direction). Clearly the changes and uncertainties in y and z are correlated — we do not expect massive changes in height for little change

[4] technically it never increases it

[5] which will always be positive semi definite

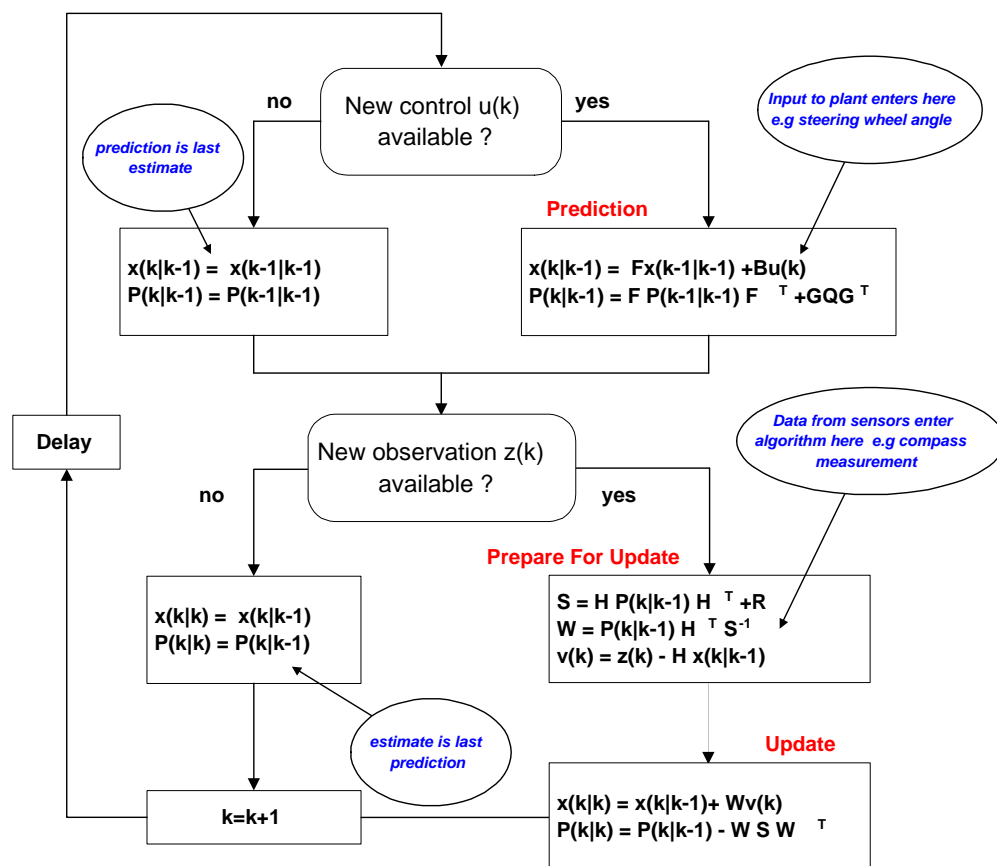



Figure 5.1: Flow chart for the Kalman filter. Note the recursive structure and how simple it is to implement once the model matrices **F** and **H** have been specified along with the model uncertainty matrices **R** and **Q** .

in distance-over-ground in normal flight conditions. Now comes the clever part. The plane is equipped with an altimeter radar which measures height above the ground - a direct measurement of z. Fusing this measurement in the Kalman filter will result in a change in estimated height and *also* a change in y-position. The reason being that the correlations between height and y-position maintained in the covariance matrix mean that changes in estimated height should imply changes in estimated y-position because the two states are co-dependent. The exercises associated with these lectures should illustrate this fact to you further.

5.2 Using Estimation Theory in Mobile Robotics

Previous pages have contained a fair amount of maths taking you through the derivation of various estimators in a steady fashion. Remember estimation theory is at the core of many problems in mobile robotics — sensors are uncertain, models are always incomplete. Tools like the Kalman filter [6] give a powerful framework in which we can progress.

We will now apply these freshly derived techniques to problems in mobile robotics focussing particularly on navigation. The following sections are not just a stand-alone-example. Within the analysis, implementation and discussion a few new ideas are introduced which will be used later in the course.

5.2.1 A Linear Navigation Problem - "Mars Lander"

A Lander is at an altitude $\mathbf{x}$ above the planet and uses a time of flight radar to detect altitude—see Figure 5.2.1. The onboard flight controller needs estimates of both height and velocity to actuate control surfaces and time rocket burns. The task is to estimate both altitude and descent velocity using only the radar. We begin modelling by assuming that the vehicle has reached its terminal velocity (but this velocity may change with height slowly). A simple model would be:

$$\mathbf{x}(k) = \underbrace{\begin{bmatrix} 1 & \delta T \\ 0 & 1 \end{bmatrix}}_{\mathbf{F}} \mathbf{x}(k-1) + \underbrace{\begin{bmatrix} \frac{\delta T^2}{2} \\ \delta T \end{bmatrix}}_{\mathbf{G}} \mathbf{v}(k) \tag{5.58}$$

where δT is the time between epochs and the state vector $\mathbf{x}$ is composed of two elements altitude and rate of altitude change (velocity)

$$\mathbf{x}(k) = \begin{bmatrix} h \\ \dot{h} \end{bmatrix}. \tag{5.59}$$

The process noise vector is a scalar, gaussian process with covariance $\mathbf{Q}$. It represents noise in acceleration (hence the quadratic time dependence when adding to a position-state).

Now we need to write down the observation equations. Think of the observation model as **"explaining the observations as a function of the thing you want to estimate"**.

[6] it is not the only method though

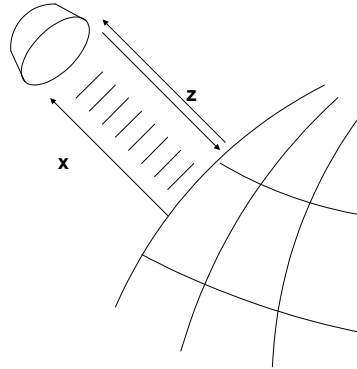



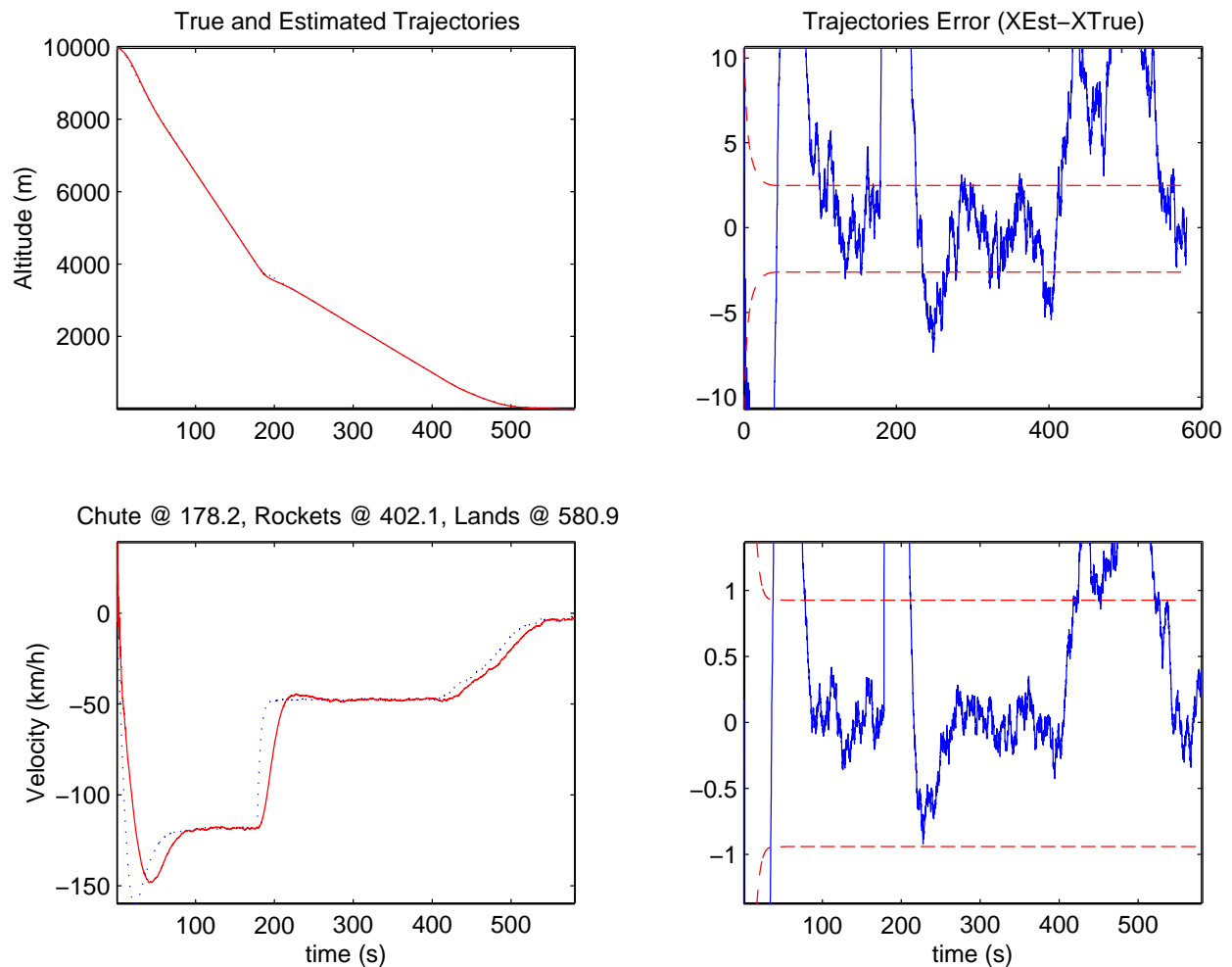
Figure 5.2: A simple linear navigation problem. A Lander is at an altitude **x** above the planet and uses a time of flight radar to detect altitude. The onboard flight controller needs estimates of both height and velocity to actuate control surfaces and time rocket burns. The task is to estimate both altitude and descent velocity using only the radar.

So we can write

$$\mathbf{z}(k) = \mathbf{H}\mathbf{x}(k) + \mathbf{w}(k) \tag{5.60}$$

$$\mathbf{z}(k) = \begin{bmatrix} \frac{2}{c} & 0 \end{bmatrix} \begin{bmatrix} h \\ \dot{h} \end{bmatrix} + \mathbf{w}(k) \tag{5.61}$$

These are the "truth models" of the system (note there are no "hats" or (k—k) notations involved). We will never known the actual value of the noises at any epoch but we model the imperfections in the sensor and motions models that they represent by using their covariance matrices (**R** and **Q** respectively) in the filter equations.

We are now in a position to implement this example in Matlab. Section 11.1 is a print out of the entire source code for a solution - it can also be downloaded from the course website *¡http://www.robots.ox.ac.uk/∼pnewman : teaching¿*. The exact source created the following graphs and so all parameters can be read from the listing. It's important that you can reconcile all the estimation equations we have derived with the source and also that you understand the structure of the algorithm —the interaction of simulator, controller and estimator.

5.2.2 Simulation Model

The simulation model is non-linear. This of course is realistic — we may *model* the world within our filter as a well behaved linear system [7] but the physics of the real world can be relied upon to conspire to yield something far more complicated. However, the details of the models employed in the simulation are not important. One point of this example is to illustrate that sometimes we can get away with simple approximations to complex systems. We only examine the vertical descent velocity (i.e. we ignore the coriolis acceleration)—it's as though we were dropping a weight vertically into a layer of atmosphere. The drag exerted on the vehicle by the atmosphere is a function of both vehicle form-factor, velocity and atmospheric density as a function of height (modelled as a saturating exponential).

[7] we shall shortly introduce a way to use non-linear models but the idea that the world is always more complicated than we care to imagine is still valid

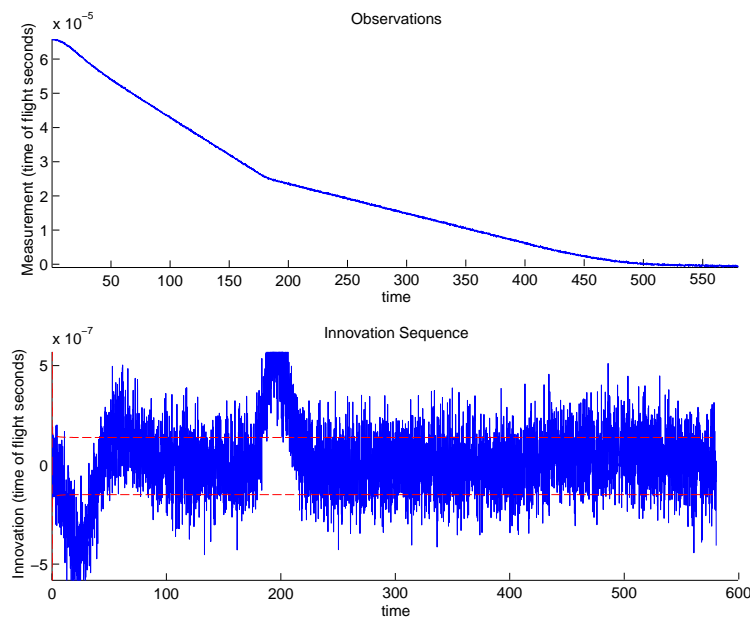



Figure 5.3:

Vehicle Controller

The controller implemented does two things. Firstly when the vehicle descends below a predetermined height it deploys a parachute which increases its effective aerodynamic drag. Secondly, it fires retro-burners when it drops below a second altitude threshold. A simple control law based on altitude and velocity is used to control the vehicle to touch down with a low velocity. At the point when rockets are fired, the parachute is also jettisoned.

The important point here is that the controller operates on estimated quantities. If the estimated quantities are in error it is quite easy to envision bad things happening. This is a point common to all robotic systems—actions (involving substantial energies) are frequently executed on the basis estimates. The motivation to understand estimation process and its failure modes is clear!

Analysis of Mars Lander Simulation

Flight Pattern Figure 5.2.1 shows the simulated(true) and estimated states using the code listed in Section 11.1. Initially the vehicle is high in thin atmosphere which produces little drag. The vehicle accelerates through the high levels of the atmosphere. Soon the density increases and the vehicle brakes under the effect of drag to reach a quasi-steady

state terminal velocity. When the parachute opens the instantaneous increase drag decelerates the vehicle rapidly until another steady state terminal velocity is achieved. Finally shortly before landing the retro-burners are fired to achieve a smooth landing - essentially decelerating the vehicle smoothly until touch down.

Fitness of Model The filter vehicle model $< \mathbf{F}, \mathbf{G} >$ is one of constant velocity (noise in acceleration). Off-the-blocks then we should expect good estimation performance during periods of constant velocity. Examining the graphs we see this is true. When the velocity is constant both position and velocity are tracked well. However during periods of rapid acceleration we see poor estimates of velocity emerging. Note that during these times the innovation sequence and truth-estimated graphs 'spike'....

Derivation of Velocity Note that the observation model $\mathbf{H}$ does not involve the velocity state and yet the filter is clearly estimating and tracking velocity pretty well. At no point in the code can you find statements like $vel = (xnew - xold)/\Delta T$. The filter is not explicitly differentiating position to derive velocity — instead it is inferring it through the models provided. The mechanism behind this has already been discussed in section 5.1.3. The filter is using correlations (off diagonals) in the 2×2 matrix $\mathbf{P}$ between position and velocity states. The covariance matrix starts off being diagonal but during the first prediction step it becomes fully populated. [8]. Errors in position are correlated through the vehicle model to errors in velocity. This is easy to spot in the plant model as predicted position is a function of current position estimate *and* velocity estimate. Here the KF is working as an observer of a hidden state — an immensely useful characteristic. However there is no free lunch. Note how during times of acceleration the velocity estimate lags behind the true velocity. This makes sense [9] the velocity state is being dragged (and hence lags) through state space by the correlations to directly observed position.

Filter Tuning An obvious question to ask is how can the filter be made "tighter"? How can we produce a more agile tracker of velocity? The answer lies in part with the process noise strength $\mathbf{Q}$. The addition of $\mathbf{GQG}^T$ at each time step dilutes the interstate correlations. By making $\mathbf{Q}$ smaller we maintain stronger correlations and track inferred velocity. But we cannot reduce $\mathbf{Q}$ too far— it has to model the uncertainty in our model. If we reduce it too much we will have too much confidence in our predictions and the update stage will have little corrective effect. The process of choosing a suitable $\mathbf{Q}$ (and $\mathbf{R}$) is called tuning. It is an important part of KF deployment and can be hard to do in practice. Fortunately there are a few concepts that can help in this process. Their derivation is more suited for a course in stochastic estimation rather than mobile

[8] You should download the code and check this out for yourself. Try forcing the off diagonals to zero and see the effect

[9] expand the KF update equations with a constant velocity model and full $\mathbf{P}$ matrix. Note how the change in velocity $\mathbf{W}(2,1)$ is a function of the off diagonals

robotics and so some of them are stated here as a set of rules (valid for linear Gaussian problems);

Innovation Sequence The innovation sequence should be a white (utterly random),zero mean process. It is helpful to think heuristically of the innovation as the exhaust from the filter. If the filter is optimal all information will have been extracted from previous observations and the difference between actual and predicted observations will be noise[10].

S-Bound The innovation covariance matrix $\mathbf{S}$ provides a $1\sigma^2$ statistical bound on the innovations sequence. The *ith* element innovation sequence should lie with $+/-\sqrt{S_{ii}}$. Figure 5.2.2 shows these bounds plotted. Note how during periods of constant velocity the innovation sequence (a scalar sequence in this case) is bounded around 66 percent of the time.

Normalised Innovation Squared The scalar quantity $\nu(k)^T\mathbf{S}^{-1}\nu(k)$ is distributed according to chi-squared distribution with degree of freedom equal to the dimension of the innovation vector $\nu(k)$

Estimate Error In a deployed system the only information available to the engineer is the innovation sequence (see above). However if a simulation is available comparisons between estimated and nominal true states can be made. The filter should be unbiased and so the average error should be zero.

Normalised Estimate Error Squared If we denote the error at epoch k between true and estimated states as $\tilde{\mathbf{x}}(k)$ then the quantity $\mathcal{E}\{\tilde{\mathbf{x}}(k)^T\mathbf{P}(k|k)^{-1}\tilde{\mathbf{x}}(k)\}$ is distributed according to chi-squared distribution with degrees of freedom equal to the state dimension.

There is some skill involved in choosing values of $\mathbf{R}$ and $\mathbf{Q}$ such that the above criteria are met, especially when the filter models are a poor representation of the truth. The correct thing to do here is implement a better model. If however, other engineering issues impede this course of action, the filter must be de-tuned (increase noise strengths) in the hope of 'lumping' un-modelled characteristics into the noise vector. This of course means that the filter looses any claim to optimality.

[10]There is a powerful geometric interpretation of the Kalman filter that fits closely to this analogy using ideas of orthogonality

5.3 Incorporating Non-Linear Models - The Extended Kalman Filter

Up until now we have only considered linear models (although working in non-linear simulated environments). It shouldn't come as a surprise to you that the majority of real world applications require the use of non-linear models. Think about an everyday example - a really simple GPS receiver sitting at $\mathbf{x}_r = (x, y, z)^T$ and measuring time of flight (equivalent) to a satellite sitting at $\mathbf{x}_s = (x, y, z)^T_s$. Clearly the time of flight measurement is "explained" by the norm of the difference vector $|| \mathbf{x}_r - \mathbf{x}_s$. This then requires the use of a non-linear measurement model. Fortunately we shall see that non-linear models can easily be incorporated into the Kalman Filter equations (yielding a filter called the Extended Kalman Filter or EKF) you are already familiar with. The derivations are given here for completeness and the results are stated in Section 5.3.3.

5.3.1 Non-linear Prediction

We begin by writing a general form for a non-linear plant truth model:

$$\mathbf{x}(k) = \mathbf{f}(\mathbf{x}(k-1), \mathbf{u}(k), k) + \mathbf{v}(k) \tag{5.62}$$

$$\mathbf{z}(k) = \mathbf{h}(\mathbf{x}(k), \mathbf{u}(k), k) + \mathbf{w}(k). \tag{5.63}$$

The trick behind the EKF is to linearize the non-linear models around the "best" current estimate (best meaning prediction $(k|k-1)$ or last estimate $(k-1|k-1)$). This is done using a Taylor series expansion. Assume we have an estimate $\hat{\mathbf{x}}(k-1|k-1)$ then

$$\mathbf{x}(k) = \mathbf{f}(\hat{\mathbf{x}}(k-1|k-1), \mathbf{u}(k), k) + \nabla\mathbf{F_x}[\mathbf{x}(k-1) - \hat{\mathbf{x}}(k-1|k-1)] + \cdots \tag{5.64}$$

The term $\nabla\mathbf{F_x}$ is understood to be the jacobian of $(\mathbf{f})$ with respect to $\mathbf{x}$ evaluated at an elsewhere specified point:

$$\nabla\mathbf{F_x} = \frac{\partial\mathbf{f}}{\partial\mathbf{x}} = \begin{bmatrix} \frac{\partial\mathbf{f}_1}{\partial\mathbf{x}_1} & \cdots & \frac{\partial\mathbf{f}_1}{\partial\mathbf{x}_m} \\ \vdots & & \vdots \\ \frac{\partial\mathbf{f}_n}{\partial\mathbf{x}_1} & \cdots & \frac{\partial\mathbf{f}_n}{\partial\mathbf{x}_m} \end{bmatrix}$$

Now we know that $\hat{\mathbf{x}}(k|k-1) = \mathcal{E}\{\mathbf{x}(k)|\mathbf{Z}^{k-1}\}$ and $\hat{\mathbf{x}}(k-1|k-1) = \mathcal{E}\{\mathbf{x}(k-1)|\mathbf{Z}^{k-1}\}$so

$$\hat{\mathbf{x}}(k|k-1) = \mathcal{E}\{\mathbf{f}(\hat{\mathbf{x}}(k-1|k-1), \mathbf{u}(k), k) + \nabla\mathbf{F_x}[\mathbf{x}(k-1) - \hat{\mathbf{x}}(k-1|k-1)] + \cdots + \mathbf{v}(k)|\mathbf{Z}^{k-1}\} \tag{5.65}$$

$$= \mathcal{E}\{\mathbf{f}(\hat{\mathbf{x}}(k-1|k-1), \mathbf{u}(k), k)|\mathbf{Z}^{k-1}\} + \nabla\mathbf{F_x}[\hat{\mathbf{x}}(k-1|k-1) - \hat{\mathbf{x}}(k-1|k-1)] \tag{5.66}$$

$$= \mathbf{f}(\hat{\mathbf{x}}(k-1|k-1), \mathbf{u}(k), k) \tag{5.67}$$

Which is an obvious conclusion — simply pass the last estimate through the non-linear model to come up with a prediction based on a control signal $\mathbf{u}$ (k). To figure out how to propagate the covariance (using only terms from the previous time step) we look at the behaviour of $\tilde{\mathbf{x}}(k|k-1) = \mathbf{x}(k) - \hat{\mathbf{x}}(k|k-1)$ because $\mathbf{P}(k|k-1) = \mathcal{E}\{\tilde{\mathbf{x}}(k|k-1)\tilde{\mathbf{x}}(k|k-1)^T|\mathbf{Z}^{k-1}\}$. Understanding that the jacobians of $\mathbf{f}$ are evaluated at $\hat{\mathbf{x}}(k-1|k-1)$ we can write

$$\tilde{\mathbf{x}}(k|k-1) = \mathbf{x}(k) - \hat{\mathbf{x}}(k|k-1) \tag{5.68}$$

$$\approx \mathbf{f}(\hat{\mathbf{x}}(k-1|k-1), \mathbf{u}(k), k) + \nabla\mathbf{F_x}[\mathbf{x}(k-1) - \hat{\mathbf{x}}(k-1|k-1)] + \mathbf{v}(k) - \tag{5.69}$$

$$\mathbf{f}(\hat{\mathbf{x}}(k-1|k-1), \mathbf{u}(k), k) \tag{5.70}$$

$$= \nabla\mathbf{F_x}[\mathbf{x}(k-1) - \hat{\mathbf{x}}(k-1|k-1)] + \mathbf{v}(k) \tag{5.71}$$

$$= \nabla\mathbf{F_x}[\tilde{\mathbf{x}}(k-1|k-1)] + \mathbf{v}(k) \tag{5.72}$$

$$\tag{5.73}$$

So

$$\mathbf{P}(k|k-1) = \mathcal{E}\{\tilde{\mathbf{x}}(k|k-1)\tilde{\mathbf{x}}(k|k-1)^T|\mathbf{Z}^{k-1}\} \tag{5.74}$$

$$\approx \mathcal{E}\{(\nabla\mathbf{F_x}\tilde{\mathbf{x}}(k-1|k-1) + \mathbf{v}(k))(\nabla\mathbf{F_x}\tilde{\mathbf{x}}(k-1|k-1) + \mathbf{v}(k))^T|\mathbf{Z}^{k-1}\} \tag{5.75}$$

$$= \mathcal{E}\{\nabla\mathbf{F_x}\tilde{\mathbf{x}}(k-1|k-1)\tilde{\mathbf{x}}(k-1|k-1)^T\nabla\mathbf{F_x}{}^T|\mathbf{Z}^{k-1}\} + \mathcal{E}\{\mathbf{v}(k)\mathbf{v}(k)^T|\mathbf{Z}^{k-1}\} \tag{5.76}$$

$$= \nabla\mathbf{F_x}\mathbf{P}(k-1|k-1)\nabla\mathbf{F_x}{}^T + \mathbf{Q} \tag{5.77}$$

We now have the predict equations in the case of non-linear plant models. Note that frequently the model will be in the form

$$\mathbf{x}(k) = \mathbf{f}(\mathbf{x}(k-1), \mathbf{u}(k), \mathbf{v}(k), k) \tag{5.78}$$

where the noise $\mathbf{v}$ (k) is not simply additive. In this case one would proceed with a multivariate Taylor[11] series which swiftly becomes notationally complex and algebraically tiresome. However the end result is intuitive. The state prediction remains unchanged but the prediction equation becomes:

$$\mathbf{P}(k|k-1) = \nabla\mathbf{F_x}\mathbf{P}(k-1|k-1)\nabla\mathbf{F_x}{}^T + \nabla\mathbf{G_v}\mathbf{Q}\nabla\mathbf{G_v}{}^T \tag{5.79}$$

[11] alternatively stack $\mathbf{x}$ and $\mathbf{v}$ in a new vector $\mathbf{y}$ and differentiate with respect to $\mathbf{y}$ - the same result follows.

where $\nabla\mathbf{F_u}$ is the jacobian of $\mathbf{f}$ w.r.t the noise vector. (Don't worry many examples to follow — also look at the source code provided) It is a common practice to make the substitution $\mathbf{u}(k) = \mathbf{u}_n(k) + \mathbf{v}(k)$ where $\mathbf{u}_n(k)$ is a nominal control which is then corrupted by noise. In this case $\nabla\mathbf{G_v} = \nabla\mathbf{G_u}$. You will see this again soon. In the meantime, look at the example source code provided.

5.3.2 Non-linear Observation Model

Now we turn to the case of a non-linear observation model (for example a range and bearing sensor) of the general form

$$\mathbf{z}(k) = \mathbf{h}(\mathbf{x}(k)) + \mathbf{w}(k) \tag{5.80}$$

By using the same technique used for the non-linear prediction step we can show that the predicted observation $\mathbf{z}(k|k-1)$ ($\mathbf{H}\hat{\mathbf{x}}(k|k-1)$ in the linear case) is simply the projection of $\hat{\mathbf{x}}(k|k-1)$ through the measurement model:

$$\mathbf{z}(k|k-1) = \mathcal{E}\{\mathbf{z}(k)|\mathbf{Z}^{k-1}\} \tag{5.81}$$

$$\mathbf{z}(k|k-1) = \mathbf{h}(\hat{\mathbf{x}}(k|k-1)). \tag{5.82}$$

Now we wish to derive an expression for $\mathbf{S}$, the innovation covariance. We begin by expressing the observation and the estimated observation error $\tilde{\mathbf{z}}(k|k-1)$ using a Taylor series:

$$\mathbf{z}(k) \approx \mathbf{h}(\hat{\mathbf{x}}(k|k-1)) + \nabla\mathbf{H_x}(\hat{\mathbf{x}}(k|k-1) - \mathbf{x}(k)) + \cdots + \mathbf{w}(k) \tag{5.83}$$

$$\tilde{\mathbf{z}}(k|k-1) = \mathbf{z}(k) - \mathbf{z}(k|k-1) \tag{5.84}$$

$$= \mathbf{h}(\hat{\mathbf{x}}(k|k-1)) + \nabla\mathbf{H_x}(\hat{\mathbf{x}}(k|k-1) - \mathbf{x}(k)) + \cdots + \mathbf{w}(k) - \mathbf{h}(\hat{\mathbf{x}}(k|k-1)) \tag{5.85}$$

$$= \nabla\mathbf{H_x}(\hat{\mathbf{x}}(k|k-1) - \mathbf{x}(k)) + \mathbf{w}(k) \tag{5.86}$$

$$= \nabla\mathbf{H_x}(\tilde{\mathbf{x}}(k|k-1)) + \mathbf{w}(k) \tag{5.87}$$

So the covariance of the difference between actual and predicted observations (the innovation) can be written as:

$$\mathbf{S} = \mathcal{E}\{\tilde{\mathbf{z}}(k|k-1)\tilde{\mathbf{z}}(k|k-1)^T|\mathbf{Z}^{k-1}\} \tag{5.88}$$

$$= \mathcal{E}\{(\nabla\mathbf{H_x}(\tilde{\mathbf{x}}(k|k-1)) + \mathbf{w}(k))(\nabla\mathbf{H_x}(\tilde{\mathbf{x}}(k|k-1)) + \mathbf{w}(k))^T|\mathbf{Z}^{k-1}\} \tag{5.89}$$

$$= \nabla\mathbf{H_x}\mathcal{E}\{\tilde{\mathbf{x}}(k|k-1)\tilde{\mathbf{x}}(k|k-1)^T|\mathbf{Z}^{k-1}\}\nabla\mathbf{H_x}^T + \mathcal{E}\{\mathbf{w}(k)\mathbf{w}(k)^T|\mathbf{Z}^{k-1}\} \tag{5.90}$$

$$= \nabla\mathbf{H_x}\mathbf{P}(k|k-1)\nabla\mathbf{H_x}^T + \mathbf{R} \tag{5.91}$$

You may be wondering where the $\mathcal{E}\{\tilde{\mathbf{x}}(k|k-1)\mathbf{w}(k)^T|\mathbf{Z}^{k-1}\}$ terms have gone. They evaluate to zero as (reasonably) we do not expect the noise in observations to be correlated to the error in our prediction.

We now have one last thing to figure out — how does a non-linear observation model change the update equations resulting in $\hat{\mathbf{x}}(k|k)$ and $\mathbf{P}(k|k)$? The procedure should now be becoming familiar to you: figure out an expression using a series expansion for $\tilde{\mathbf{x}}(k|k)$ and take squared conditional expectation to evaluate $\mathbf{P}(k|k)$.

Assume that "somehow" we have a gain $\mathbf{W}$ (we'll derive this in a minute) then we can immediately write:

$$\hat{\mathbf{x}}(k|k) = \underbrace{\hat{\mathbf{x}}(k|k-1)}_{prediction} + \underbrace{\mathbf{W}}_{gain} \underbrace{(\underbrace{\mathbf{z}(k)}_{observation} - \underbrace{\mathbf{h}(\hat{\mathbf{x}}(k|k-1))}_{predictedobservation}}_{innovation} \tag{5.92}$$

We can now use this expression to make progress in figuring out $\mathbf{P}(k|k)$.

$$\mathbf{P}(k|k) = \mathcal{E}\{\tilde{\mathbf{x}}(k|k)\tilde{\mathbf{x}}(k|k)^T|\mathbf{Z^k}\} \tag{5.93}$$

$$\tilde{\mathbf{x}}(k|k) = \hat{\mathbf{x}}(k|k) - \mathbf{x}(k) \tag{5.94}$$

$$= \hat{\mathbf{x}}(k|k-1) + \mathbf{W}(\mathbf{z}(k) - \mathbf{h}(\hat{\mathbf{x}}(k|k-1))) - \mathbf{x}(k) \tag{5.95}$$

$$= \hat{\mathbf{x}}(k|k-1) + \mathbf{W}\tilde{\mathbf{z}}(k|k-1) - \mathbf{x}(k) \tag{5.96}$$

and substituting equation 5.87

$$= \hat{\mathbf{x}}(k|k-1) + \mathbf{W}(\nabla\mathbf{H_x}(\tilde{\mathbf{x}}(k|k-1)) + \mathbf{w}(k)) - \mathbf{x}(k) \tag{5.97}$$

$$= \tilde{\mathbf{x}}(k|k-1) + \mathbf{W}\nabla\mathbf{H_x}\tilde{\mathbf{x}}(k|k-1) + \mathbf{W}\mathbf{w}(k) \tag{5.98}$$

$$= [\mathbf{I} - \mathbf{W}\nabla\mathbf{H_x}]\tilde{\mathbf{x}}(k|k-1) + \mathbf{W}\mathbf{w}(k) \tag{5.99}$$

An expression for $\mathbf{P}(k|k)$ follows swiftly as

$$\mathbf{P}(k|k) = \mathcal{E}\{\tilde{\mathbf{x}}(k|k-1)\tilde{\mathbf{x}}(k|k-1)^T|\mathbf{Z^k}\} \tag{5.100}$$

$$= \mathcal{E}\{([\mathbf{I} - \mathbf{W}\nabla\mathbf{H_x}]\tilde{\mathbf{x}}(k|k-1) + \mathbf{W}\mathbf{w}(k))([\mathbf{I} - \mathbf{W}\nabla\mathbf{H_x}]\tilde{\mathbf{x}}(k|k-1) + \mathbf{W}\mathbf{w}(k))^T|\mathbf{Z^k}\} \tag{5.101}$$

$$= [\mathbf{I} - \mathbf{W}\nabla\mathbf{H_x}]\mathcal{E}\{\tilde{\mathbf{x}}(k|k-1)\tilde{\mathbf{x}}(k|k-1)^T|\mathbf{Z^k}\}[\mathbf{I} - \mathbf{W}\nabla\mathbf{H_x}]^T + \mathbf{W}\mathcal{E}\{\mathbf{w}(k)\mathbf{w}(k)^T|\mathbf{Z^k}\}\mathbf{W}^T \tag{5.102}$$

$$= [\mathbf{I} - \mathbf{W}\nabla\mathbf{H_x}]\mathbf{P}(k|k-1)[\mathbf{I} - \mathbf{W}\nabla\mathbf{H_x}]^T + \mathbf{W}\mathbf{R}\mathbf{W}^T \tag{5.103}$$

Above, we have used the fact that the expectation of $\tilde{\mathbf{x}}$ is zero and so the cross-terms of the expansion evaluate to zero. We now just need to find an expression for $\mathbf{W}$. Here we derive the gain using a little calculus. Recall that behind all of this is a quest to minimise the mean square estimation error (which is scalar):

$$\mathbf{J}(\hat{\mathbf{x}}) = \mathcal{E}\{\tilde{\mathbf{x}}(k|k)^T\tilde{\mathbf{x}}(k|k)|kZk\} \tag{5.104}$$

$$= Tr(\mathbf{P}(k|k)) \tag{5.105}$$

Fortunately there is a closed form to the differential of such a function. If **B** is symmetric for any **A** then

$$\frac{\partial Tr(\mathbf{ABA}^T)}{\partial \mathbf{A}} = 2\mathbf{AB} \quad \frac{\partial \mathbf{AC}}{\partial \mathbf{A}} = \mathbf{C}^T \tag{5.106}$$

Applying this (and the chain rule) to Equation 5.103 and setting the derivative equal to zero we get:

$$\frac{\partial \mathbf{J}(\hat{\mathbf{x}})}{\partial \mathbf{W}} = -2[\mathbf{I} - \mathbf{W}\nabla\mathbf{H_x}]\mathbf{P}(k|k-1)\nabla\mathbf{H_x}^T + 2\mathbf{WR} = 0 \tag{5.107}$$

$$\mathbf{W} = \mathbf{P}(k|k-1)\nabla\mathbf{H_x}^T(\nabla\mathbf{H_x}\mathbf{P}(k|k-1)\nabla\mathbf{H_x}^T + \mathbf{R})^{-1} \quad = \mathbf{P}(k|k-1)\nabla\mathbf{H_x}^T\mathbf{S}^{-1} \tag{5.108}$$

where

$$\mathbf{S} = (\nabla\mathbf{H_x}\mathbf{P}(k|k-1)\nabla\mathbf{H_x}^T + \mathbf{R}) \tag{5.109}$$

Note that substituting $\mathbf{P}(k|k-1)\nabla\mathbf{H_x}^T = \mathbf{WS}$ into Equation 5.103 leads to the form of the covariance update we are already familiar with:

$$\mathbf{P}(k|k) = \mathbf{P}(k|k-1) - \mathbf{WSW}^T \tag{5.110}$$

5.3.3 The Extended Kalman Filter Equations

We can now state in a "good to remember this box" a rule for converting the linear Kalman filter equations to the non-linear form:

> To convert the linear Kalman Filter to the Extended Kalman Filter simply replace **F** with $\nabla\mathbf{F_x}$ and **H** with $\nabla\mathbf{H_x}$ **in the covariance and gain calculations only**. The jacobians are always evaluated at the best available estimate ($\hat{\mathbf{x}}(k-1|k-1)$ for $\nabla\mathbf{F_x}$ and $\hat{\mathbf{x}}(k|k-1)$ for $\nabla\mathbf{H_x}$

.

For completeness here are the EKF equations. (You'll need these for the class-work). If you don't feel you are on top of the previous maths - its not the end of the world. Make

sure however you are familiar with the general form of the equations. (exam useful things are in boxes in these notes (but not exclusively))

Prediction:

$$\underbrace{\hat{\mathbf{x}}(k|k-1)}_{\text{predicted state}} = \overbrace{\mathbf{f}(\underbrace{\hat{\mathbf{x}}(k-1|k-1)}_{\text{old state est}}, \underbrace{\mathbf{u}(k)}_{\text{control}}, k)}^{\text{plant model}} \tag{5.111}$$

$$\underbrace{\mathbf{P}(k|k-1)}_{\text{predicted covariance}} = \nabla\mathbf{F_x}\underbrace{\mathbf{P}(k-1|k-1)}_{\text{old est covariance}}\nabla\mathbf{F_x}^T + \underbrace{\nabla\mathbf{G_v}\mathbf{Q}\nabla\mathbf{G_v}^T}_{\text{process noise}} \tag{5.112}$$

$$\underbrace{\mathbf{z}(k|k-1)}_{\text{predicted obs}} = \overbrace{\mathbf{h}(\hat{\mathbf{x}}(k|k-1))}^{\text{observation model}} \tag{5.113}$$

Update:

$$\underbrace{\hat{\mathbf{x}}(k|k)}_{\text{new state estimate}} = \overbrace{\hat{\mathbf{x}}(k|k-1) + \mathbf{W}\underbrace{\nu(k)}_{\text{innovation}}}^{\text{prediction and correction}} \tag{5.114}$$

$$\underbrace{\mathbf{P}(k|k)}_{\text{new covariance estimate}} = \underbrace{\mathbf{P}(k|k-1) - \mathbf{WSW}^T}_{\text{update decreases uncertainty}} \tag{5.115}$$

where

$$\nu(k) = \overbrace{\mathbf{z}(k)}^{\text{measurement}} - \mathbf{z}(k|k-1) \tag{5.116}$$

$$\mathbf{W} = \underbrace{\mathbf{P}(k|k-1)\nabla\mathbf{H_x}^T\mathbf{S}^{-1}}_{\text{kalman gain}} \tag{5.117}$$

$$\mathbf{S} = \underbrace{\nabla\mathbf{H_x}\mathbf{P}(k|k-1)\nabla\mathbf{H_x}^T + \mathbf{R}}_{\text{Innovation Covariance}} \tag{5.118}$$

$$\nabla\mathbf{F_x} = \frac{\partial\mathbf{f}}{\partial\mathbf{x}} = \underbrace{\begin{bmatrix} \frac{\partial\mathbf{f}_1}{\partial\mathbf{x}_1} & \cdots & \frac{\partial\mathbf{f}_1}{\partial\mathbf{x}_m} \\ \vdots & & \vdots \\ \frac{\partial\mathbf{f}_n}{\partial\mathbf{x}_1} & \cdots & \frac{\partial\mathbf{f}_n}{\partial\mathbf{x}_m} \end{bmatrix}}_{\text{evaluated at } \hat{\mathbf{x}}(k-1|k-1)} \nabla\mathbf{H_x} = \frac{\partial\mathbf{h}}{\partial\mathbf{x}} = \underbrace{\begin{bmatrix} \frac{\partial\mathbf{f}_1}{\partial\mathbf{x}_1} & \cdots & \frac{\partial\mathbf{h}_1}{\partial\mathbf{x}_m} \\ \vdots & & \vdots \\ \frac{\partial\mathbf{h}_n}{\partial\mathbf{x}_1} & \cdots & \frac{\partial\mathbf{h}_n}{\partial\mathbf{x}_m} \end{bmatrix}}_{\text{evaluated at } \hat{\mathbf{x}}(k|k-1)} \tag{5.119}$$

We are now in a great position, possessing some very powerful tools which we shall now apply to some key autonomous navigation problems.

Topic 6

Vehicle Models and Odometry

6.1 Velocity Steer Model

This is a good point to write down a simple motion model for a mobile robotic vehicle. We allow the vehicle to move on a 2D surface (a floor) and point in arbitrary directions. We parameterise the vehicle pose $\mathbf{x}_v$ (the joint of position and orientation) as

$$\mathbf{x}_v = \begin{bmatrix} x_v \\ y_v \\ \theta_v \end{bmatrix}. \tag{6.1}$$

Figure 6.1 is a diagram of a non-holonomic (local degrees of freedom less than global degree of freedom) vehicle with "Ackerman" steering. The angle of the steering wheels is given by ϕ and the instantaneous forward velocity (sometimes called throttle) is V. With reference to Figure 6.1, we immediately see that

$$\dot{x}_v = V\delta T\cos(\theta_v) \tag{6.2}$$

$$\dot{y}_v = V\delta T\sin(\theta_v) \tag{6.3}$$

$$\tag{6.4}$$

Using the instantaneous center of rotation we can calculate the rate of change of orien-

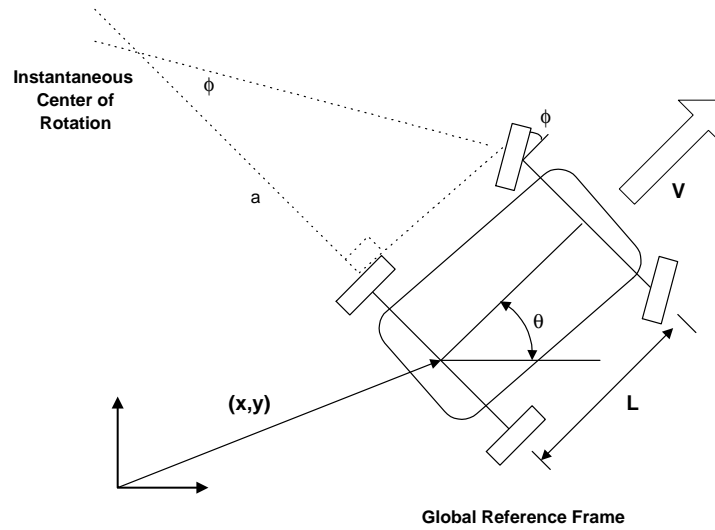



Figure 6.1: A non-holonomic vehicle with Ackerman steering

tation as a function of steer angle:

$$\frac{L}{a} = \tan(\phi) \tag{6.5}$$

$$a\dot{\theta}_v = V \tag{6.6}$$

$$\dot{\theta}_v = \frac{V}{L}\tan(\phi) \tag{6.7}$$

We can now discretise this model by inspection:

$$\mathbf{x}_v(k+1) = \mathbf{f}(\mathbf{x}_v(k), \mathbf{u}(k)); \quad \mathbf{u}(k) = \begin{bmatrix} V(k) \\ \phi(k) \end{bmatrix} \tag{6.8}$$

$$\begin{bmatrix} x_v(k+1) \\ y_v(k+1) \\ \theta_v(k+1) \end{bmatrix} = \begin{bmatrix} x_v(k+1) + \delta T V(k)\cos(\theta_v(k)) \\ y_v(k+1) + \delta T V(k)\sin(\theta_v(k)) \\ \theta_v(k) + \frac{\delta T V(k)\tan(\phi(k))}{L} \end{bmatrix} \tag{6.9}$$

Note that we have started to lump the throttle and steer into a control vector — this makes sense if you think about the controlling actions of a human driver. Equation 6.9 is a model for a perfect, noiseless vehicle. Clearly this is a little unrealistic — we need to model uncertainty. One popular way to do this is to insert noise terms into the control signal $\mathbf{u}$ such that

$$\mathbf{u}(k) = \mathbf{u}_n(k) + \mathbf{v}(k) \tag{6.10}$$

where $\mathbf{u}_n(k)$ is a nominal (intended) control signal and $\mathbf{v}(k)$ is a zero mean gaussian dis-

tributed noise vector:

$$\mathbf{v}(k) \sim \mathcal{N}(0, \begin{bmatrix} \sigma_V^2 & 0 \\ 0 & \sigma_\phi^2 \end{bmatrix}) \tag{6.11}$$

$$\mathbf{u}(k) \sim \mathcal{N}(\mathbf{u}_n(k), \begin{bmatrix} \sigma_V^2 & 0 \\ 0 & \sigma_\phi^2 \end{bmatrix}) \tag{6.12}$$

This completes a simple probabilistic model of a vehicle. We shall now see how propagation of this model affects uncertainty in vehicle pose over time.

6.2 Evolution of Uncertainty

In this section we will examine how an initial uncertainty in vehicle pose increases over time as the vehicle moves when only the control signal $\mathbf{u}$ is available. Hopefully you will realise that one way to approach this is repetitive application of the prediction step of a Kalman filter discussed in Section 5.1.1. The model derived in the previous section is non-linear and so we will have to use the non-linear form of the prediction step.

Assume at time k we have been given a previous best estimate of the vehicle pose $\hat{\mathbf{x}}_v(k-1|k-1)$ and an associated covariance $\mathbf{P}_v(k-1|k-1)$. Equations 5.67 and 5.79 have that:

$$\mathbf{x}_v(k|k-1) = \mathbf{f}(\hat{\mathbf{x}}(k-1|k-1), \mathbf{u}(k), k) \tag{6.13}$$

$$\mathbf{P}_v(k|k-1) = \nabla\mathbf{F_x}\mathbf{P}_v(k-1|k-1)\nabla\mathbf{F_x}^T + \nabla\mathbf{F_v}\mathbf{Q}\nabla\mathbf{F_v}^T \tag{6.14}$$

In this case

$$\mathbf{Q} = \begin{bmatrix} \sigma_V^2 & 0 \\ 0 & \sigma_\phi^2 \end{bmatrix} \tag{6.15}$$

We need to evaluate the Jacobians with respect to state and control noise at $\hat{\mathbf{x}}(k-1|k-1)$. We do this by differentiating each row of $\mathbf{f}$ by each state and each control respectively:

$$\nabla\mathbf{F_x} = \begin{bmatrix} 1 & 0 & -\delta T V \sin(\theta_v) \\ 0 & 1 & \delta T V \cos(\theta_v) \\ 0 & 0 & 1 \end{bmatrix} \tag{6.16}$$

$$\nabla\mathbf{F_u} = \begin{bmatrix} \delta T \cos(\theta_v) & 0 \\ \delta T \sin(\theta_v) & 0 \\ \frac{\tan(\phi)}{L} & \frac{\delta T V \sec^2(\phi)}{L} \end{bmatrix} \tag{6.17}$$

Figure 6.2 shows the results of iterating equations 6.13 and 6.14 using the matlab code printed in Section 11.2. Things are pretty much as we might expect. The uncertainty

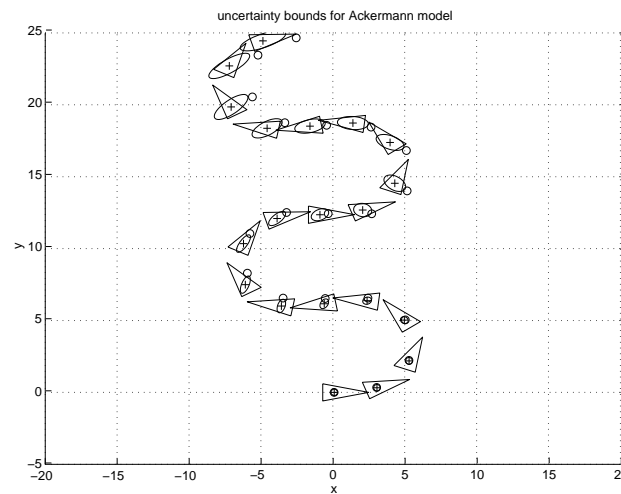



Figure 6.2: Evolution of uncertainty evolution of open loop the Ackermann Model. The circles are the true location of the vehicle whereas the crosses mark the dead-reckoned locations. The orientation of the vehicle is made clear by the orientation of the triangles. Note the divergence between true and dead-reckoned locations. This is typical of all dead reckoning methods — the only thing that can be changed is the rate of divergence.

injected into the system via the noisy control makes the estimated covariance of the vehicles grow without bound.

There is an important point to make here that you must understand. In actual real life the real robot is integrating the noisy control signal. The true trajectory will therefore *always* drift away from the trajectory estimated by the algorithms running inside the robot. This is exactly the same as closing your eyes and trying to walk across University Parks. Your inner ears and legs give you $\mathbf{u}$ which you pass through your own kinematic model of your body in your head. Of course, one would expect a gross accumulation of error as the time spent walking "open loop" increases. The point is that all measurements such as velocity and rate of turn are measured in the vehicle frame and must be integrated, along with the noise on the measurements. This always leads to what is called "dead reckoning drift". Figure 6.3 shows the effect of integrating odometry on a real robot called "B21" shown in figure 6.3(right). The main cause of this divergence on land vehicles is wheel slip. Typically robot wheels are fitted with encoders that measure the rotation of each wheel. Position is then an integral-function of these "wheel counts". The problem is a wheel or radius r may have turned through θ but due to slip/skid the distance travelled over the ground is only $(1-\eta)r\theta$ where η is an *unobservable* slip parameter.

6.3 Using Dead-Reckoned Odometry Measurements

The model in Section 6.1 used velocity and steer angles as control input into the model. It is common to find that this low level knowledge is not easy to obtain or that the relationship between control, prior and prediction is not readily discernable. The architecture in figure 6.3 is a typical example.

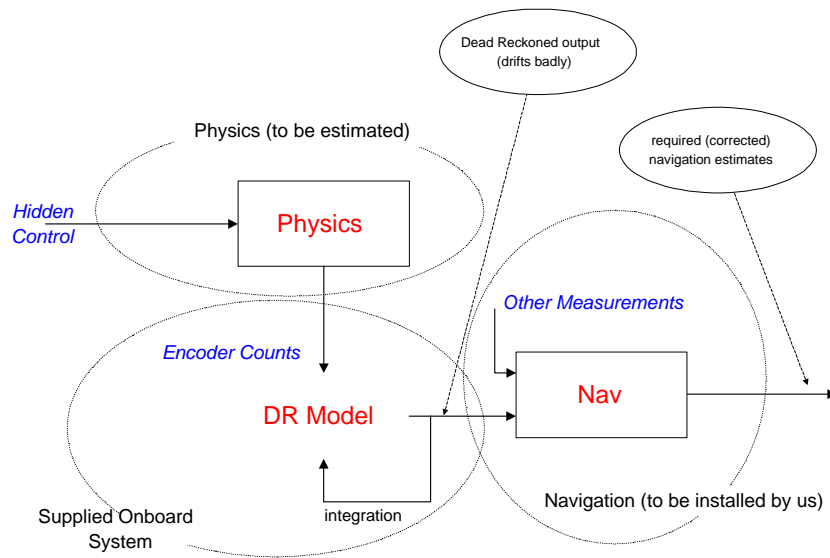



Figure 6.3: Sometimes a navigation system will be given a dead reckoned position as input without recourse to the control signals that were involved. Nevertheless the dead-reckoned position can be converted into a control input (a stream of small motions) for use in the core navigation system.

It would clearly be a bad plan to simply use a dead-reckoned odometry estimate as a direct measurement of state in something like a Kalman Filter. Consider Figure 6.3 which is the dead reckoned position of a B21 mobile robot (shown on right of figure) moving around some corridors. Clearly by the end of the experiment we cannot reasonably interpret dead-reckoned position as an unbiased measurement of position!

The low level controller on the vehicle reads encoders on the vehicle's wheels and outputs an estimate (with no metric of uncertainty) of its location. We can make a guess at the kind of model it uses[1] . Assume it has two wheels (left and right), radius r mounted either side of its center of mass which in one time interval turn an amount $\delta\theta_l, \delta\theta_r$ — as shown in Figure 6.3. We align a body-centered co-ordinate frame on the vehicle as shown. We want

[1]This for illustration only - the real b21 vehicle is actually a synchronous drive machine in which all four wheels change direction http://www.irobot.com

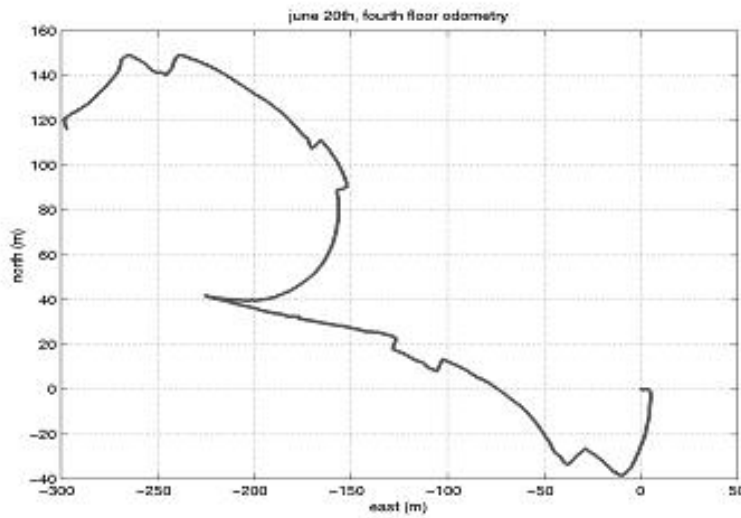



Figure 6.4: Dead Reckoned position from a B21 mobile robot. The start and end locations are actually the same place! See how you could roll the trajectory back over itself. This is typical of dead reckoned trajectories — small angular errors integrate to give massive long term errors

to express the change of position of the center of the vehicle as a function of $\delta\theta_l, \delta\theta_r$:

$$r\delta\theta_r = (c - L/2)\alpha \tag{6.18}$$

$$r\delta\theta_l = (c + L/2)\alpha \tag{6.19}$$

$$\Rightarrow c = \frac{L}{2}\frac{\delta\theta_l + \delta\theta_r}{\delta\theta_l - \delta\theta_r} \tag{6.20}$$

$$\Rightarrow \alpha = \frac{2r}{L}(\delta\theta_l - \delta\theta_r) \tag{6.21}$$

Immediately then we have

$$\begin{bmatrix} dx \\ dy \\ d\theta \end{bmatrix} = \begin{bmatrix} (1 - \cos\alpha)c \\ c\sin\alpha \\ -\alpha \end{bmatrix} \tag{6.22}$$

Which for small α becomes:

$$\begin{bmatrix} dx \\ dy \\ d\theta \end{bmatrix} = \begin{bmatrix} 0 \\ r(\delta\theta_l + \delta\theta_r)/2 \\ \frac{-2r(\delta\theta_l - \delta\theta_r)}{L} \end{bmatrix} \tag{6.23}$$

The dead-reckoning system in the vehicle simply compounds these small changes in position

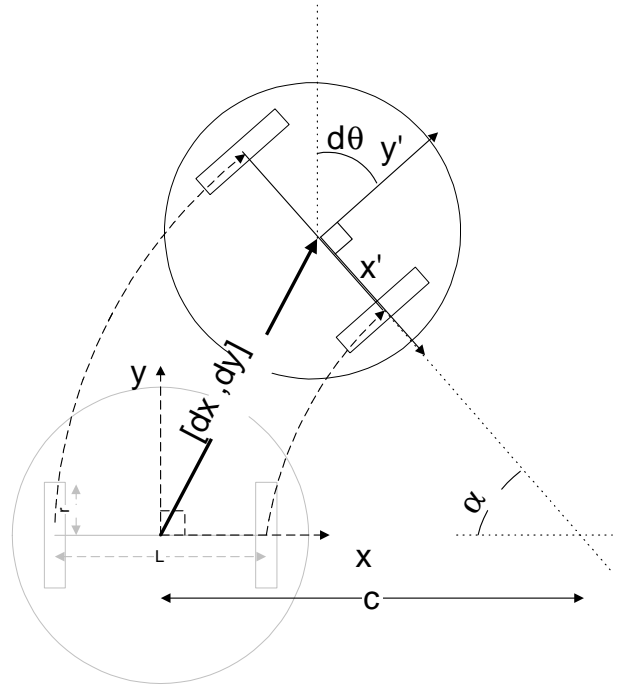



Figure 6.5: Geometric Construction for a two wheel drive vehicle

and orientation to obtain a global position estimate. Starting from an initial nominal frame at each iteration of its sensing loop it deduces a small change in position and orientation, and then "adds" this to its last dead-reckoned position. Of course the "addition" is slightly more complex than simple adding (otherwise the x coordinate would always be zero!). What actually happens is that the vehicle **composes** successive co-ordinate transformation. This is an important concept (which you will probably have met in other courses but perhaps with a different notation) and will be discussed in the next section.

6.3.1 Composition of Transformations

Figure 6.3.1 shows three relationships between three coordinate frames. We can express any coordinate j frame with respect to another frame i as a three-vector $\mathbf{x}_{i,j} = [xy\theta]$. Here x and y are translations in frame i to a point p and θ is anti-clockwise rotation around p. We define two operators $\oplus$ and $\ominus$ to allow us to compose (chain together) multiple transformations:

$$\mathbf{x}_{i,k} = \mathbf{x}_{i,j} \oplus \mathbf{x}_{j,k} \tag{6.24}$$

$$\mathbf{x}_{j,i} = \ominus \mathbf{x}_{i,j} \tag{6.25}$$

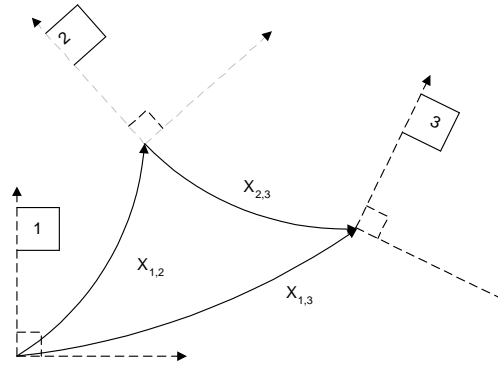



Figure 6.6: Three transformations between co-ordinate frames

With reference to figure 6.3.1 we see that $\mathbf{x}_{1,3} = \mathbf{x}_{1,2} \oplus \mathbf{x}_{2,3}$. But what *exactly* are these operators? Well, they are just a short hand for a function of one or two transformations:

$$\mathbf{x}_1 \oplus \mathbf{x}_2 = \begin{bmatrix} x_1 + x_2 \cos\theta_1 - y_2 \sin\theta_1 \\ y_1 + x_2 \sin\theta_1 + y_2 \cos\theta_1 \end{bmatrix} \tag{6.26}$$

$$\ominus \mathbf{x}_1 = \begin{bmatrix} -x_1 \cos\theta_1 - y_1 \sin\theta_1 \\ x_1 \sin\theta_1 - y_1 \cos\theta_1 \\ -\theta_1 \end{bmatrix} \tag{6.27}$$

Be sure you understand exactly what these equations allow. They allow you to express something (perhaps a point or vehicle) described in one frame, in another alternative frame. We can use this notation to explain the compounding of odometry measurements. Figure 6.3.1 shows a vehicle with a prior pose $\mathbf{x}_o(k-1)$. The processing of wheel rotations between successive readings (via Equation 6.23) has indicated a vehicle-relative transformation (i.e. in the frame of the vehicle) $\mathbf{u}$. The task of combining this new motion $\mathbf{u}(k)$ with the old dead-reckoned estimate $\mathbf{x}_o$ to arrive at a new dead-reckoned pose $\mathbf{x}_o$ is trivial. It is simply:

$$\mathbf{x}_o(k) = \mathbf{x}_o(k-1) \oplus \mathbf{u}(k) \tag{6.28}$$

We have now explained a way in which measurements of wheel rotations can be used to estimate dead-reckoned pose. However we set out to figure out a way in which a dead-reckoned pose could be used to form a control input or measurement into a navigation system. In

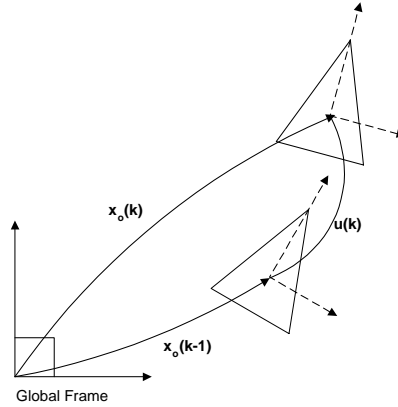



Figure 6.7: Using transformation compositions to compound a local odometry measurement with a prior dead-reckoned estimate to deduce a new dead-reckoned estimate.

other words we are given from the low-level vehicle software a sequence $\mathbf{x}_o(1), \mathbf{x}_o(2) \cdots \mathbf{x}_o(k)$ etc and we want to figure out $\mathbf{u}$ (k). This is now simple— we can invert equation 6.28 to get

$$\mathbf{u}(k) = \ominus \mathbf{x}_o(k-1) \oplus \mathbf{x}_o(k) \tag{6.29}$$

Looking at the Figure 6.3.1 we can see that the transformation $\mathbf{u}(k)$ is equivalent to going *back* along $\mathbf{x}_o(k-1)$ and forward along $\mathbf{x}_o(k)$. This gives us a small control vector $\mathbf{u}$ (k) derived from two successive dead-reckoned poses that is suitable for use in another hopefully less error prone navigation algorithm. Effectively equation 6.29 subtracts out the common dead-reckoned gross error - locally odometry is good - globally it is poor.

We are now in a position to write down a plant model for a vehicle using a dead-reckoned position as a control input:

$$\mathbf{x}_v(k+1) = \mathbf{f}(\mathbf{x}_v(k), \mathbf{u}(k)) \tag{6.30}$$

$$= \mathbf{x}_v(k) \oplus (\underbrace{\ominus \mathbf{x}_o(k-1) \oplus \mathbf{x}_o(k)}_{dr-control}) \tag{6.31}$$

$$= \mathbf{x}_v(k) \oplus \mathbf{u}_o(k) \tag{6.32}$$

It is reasonable to ask how an initial uncertainty in vehicle pose $\mathbf{P}_v$ propagates over time. We know that one way to address this question is to propagate the second order statistics (covariance) of a pdf for $\mathbf{x}_v$ through $\mathbf{f}$ using equation 5.79. To do this we need to figure out the jacobians of equation 6.32 with respect to $\mathbf{x}_v$ and $\mathbf{u}$. This is one area where the compositional representation we have adopted simplifies matters. We can define and calculate the following jacobians:

$$\mathbf{J}_1(\mathbf{x}_1, \mathbf{x}_2) \triangleq \frac{\partial(\mathbf{x}_1 \oplus \mathbf{x}_2)}{\partial \mathbf{x}1} \tag{6.33}$$

$$= \begin{bmatrix} 1 & 0 & -x_2 \sin\theta_1 - y_2 \cos\theta_1 \\ 0 & 1 & x_2 \cos\theta_1 - y_2 \sin\theta_1 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.34}$$

$$\mathbf{J}_2(\mathbf{x}_1, \mathbf{x}_2) \triangleq \frac{\partial(\mathbf{x}_1 \oplus \mathbf{x}_2)}{\partial \mathbf{x}2} \tag{6.35}$$

$$= \begin{bmatrix} \cos\theta_1 & -\sin\theta_1 & 0 \\ \sin\theta_1 & \cos\theta_1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{6.36}$$

This allows us to write (substituting into equation 5.79) :

$$\mathbf{P}(k|k-1) = \nabla\mathbf{F_x}\mathbf{P}_v(k-1|k-1)\nabla\mathbf{F_x}^T + \nabla\mathbf{F_v}\mathbf{Q}\nabla\mathbf{F_v}^T \tag{6.37}$$

$$= \mathbf{J}_1(\mathbf{x}_v, \mathbf{u}_o)\mathbf{P}_v(k-1|k-1)\mathbf{J}_1(\mathbf{x}_v, \mathbf{u}_o)^T + \mathbf{J}_2(\mathbf{x}_v, \mathbf{u}_o)\mathbf{U}_o\mathbf{J}_2(\mathbf{x}_v, \mathbf{u}_o)^T \tag{6.38}$$

Here the matrix $\mathbf{U}_o$ describes the strength of noise in the small shifts in pose represented by $\mathbf{u}_o$ derived from two sequential dead-reckoned poses. A simple form of this matrix would be purely diagonal [2]:

$$\mathbf{U}_o = \begin{bmatrix} \sigma^2_{ox} & 0 & 0 \\ 0 & \sigma^2_{oy} & 0 \\ 0 & 0 & \sigma^2_{o\theta} \end{bmatrix} \tag{6.39}$$

where the diagonals are variances in odometry noise. For example if the odometry loop ran at 20Hz and the vehicle is moving at 1m/s the magnitude of translation in $\mathbf{u}$ would be $5cm$. If we say slip accounts for perhaps one percent of distance travelled we might "try" a value of $\sigma^2_{ox} = \sigma^2_{oy} = (0.05/100)^2$. Allowing a maximum rotation of w perhaps a good starting guess for $\sigma^2_{o\theta}$ would be $(w/100)^2$. These numbers will give sensible answers while the vehicle is moving but not when it is stopped. Even when $\mathbf{u}_o = 0$ the covariance $\mathbf{P}_v$ will continue to inflate. This motivates the use of a time varying $\mathbf{U}_o$ which is a function of $\mathbf{u}_o(k)$. An exercise you should think about.....

[2] implying we expect errors in x y and orientations to be uncorrelated which is probably not true in reality but we will live with this approximation for now

Topic 7

Feature Based Mapping and Localisation

7.1 Introduction

We will now apply the estimation techniques we have learnt to two very important mobile robotics tasks - Mapping and Localisation. Theses two tasks are fundamental to successful deployment of autonomous vehicles and systems for example:

Mapping	Managing autonomous open-cast mining	GPS, Radar, Inertial
	Battlefield Surveillance	Cameras, GPS, Inertial
	Fracture detection	x-ray acoustic
	Sub-sea Oil field detection on an AUV	Acoustic Beacons, Inertial
Localisation	GPS	Satellite and time of flight
	Museum Guides	Sonar, Scanning Laser, Cameras
	Hospital Delivery System	Radio tags, laser cameras

A common way to approach these problems is to parameterise both the robot pose and aspects of the environment's geometry into one or more state vectors. For this course we

will work mostly in 2D but the definitions that follow are, of course, valid for the full 3D case.

7.2 Features and Maps

We suppose that the world is populated by a set of discrete **landmarks** or **features** whose location / orientation and geometry (with respect to a defined coordinate frame) can be described by a set of parameters which we lump into a feature vector $\mathbf{x}_f$.

We call a collection of n features a **Map** such that $\mathbf{M} = \{\mathbf{x}_{\mathbf{f},1}, \mathbf{x}_{\mathbf{f},2}, \mathbf{x}_{\mathbf{f},3} \cdots \mathbf{x}_{\mathbf{f},n}\}$. To keep notation simple sometimes we will use $\mathbf{M}$ to denote the map vector which is simply the concatenation of all the features:

$$\mathbf{M} = \begin{bmatrix} \mathbf{x}_{\mathbf{f},1} \\ \mathbf{x}_{\mathbf{f},2} \\ \vdots \\ \mathbf{x}_{\mathbf{f},n} \end{bmatrix} \tag{7.1}$$

In this course we will constrain ourselves to using the simplest feature possible - a point feature such that for the i^{th} feature:

$$\mathbf{x}_{\mathbf{f},i} = \begin{bmatrix} x_i \\ y_i \end{bmatrix} \tag{7.2}$$

where x and y are the coordinates of the point in a global frame of reference. Point features are not as uncommon as one might initially think. Points occur at the intersection of lines, corners of rectangles, edges of objects (regions of maximal curvature)[1].

7.3 Observations

We define two distinct types of observations all denoted as $\mathbf{z}$ — vehicle relative and absolute.

Absolute Absolute observations are made with the help of some external device and usually involve a direct measurement of some aspect of the vehicle's pose. The best examples

[1] Sure, our own vision system operates at a higher level recognising things like gorillas and sail-boats as complete entities but lower-level geometric-primitive detection is buried in there as well.

are a GPS and a compass. They depend on external infrastructure (taken as known) and are nothing to do with the map.

Vehicle Relative This kind of observation involves sensing the relationship between the vehicle and its immediate surroundings –especially the map, see figure 7.3. A great example is the measurement of the angle and distance to a point feature *with respect to the robot's own frame of reference*. We shall also class odometry (integration of wheel movement) as a vehicle-relative measurement because it is not a direct measurement of the vehicle's state.

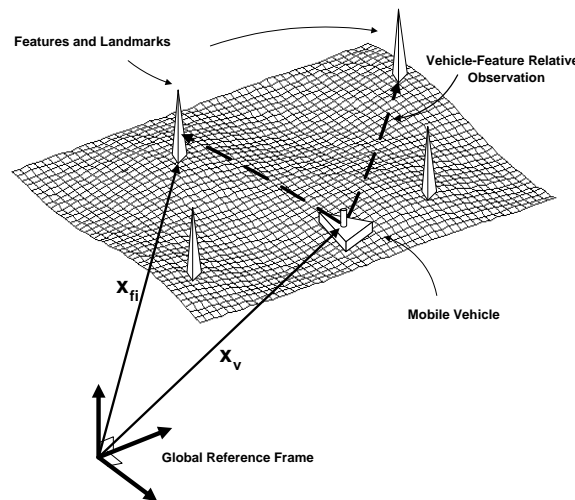



Figure 7.1: Feature Based Navigation and Mapping

7.4 A Probabilistic Framework

It is informative to describe the localisation and mapping tasks in terms of likelihoods (observation pdf) and priors.

7.4.1 Probabilistic Localisation

For the localisation task we assume we have been given a map and receive a sequence of vehicle-relative observations described by a likelihood $p(\mathbf{Z^k}|\mathbf{M}, \mathbf{x}_v)$. We wish to figure a pdf

for the vehicle pose given map and observations:

$$p(\mathbf{x}_v|\mathbf{M},\mathbf{Z^k}) = \frac{p(\mathbf{Z^k}|\mathbf{M},\mathbf{x}_v)p(\mathbf{M},\mathbf{x}_v)}{p(\mathbf{M},\mathbf{Z^k})} \tag{7.3}$$

$$= \frac{p(\mathbf{Z^k}|\mathbf{M},\mathbf{x}_v)p(\mathbf{x}_v|\mathbf{M})p(\mathbf{M})}{p(\mathbf{M},\mathbf{Z^k})} \tag{7.4}$$

$$= \frac{p(\mathbf{Z^k}|\mathbf{M},\mathbf{x}_v)p(\mathbf{x}_v|\mathbf{M})p(\mathbf{M})}{\int_{-\infty}^{\infty} p(\mathbf{Z^k}|\mathbf{M},\mathbf{x})p(\mathbf{x}_v|\mathbf{M})p(\mathbf{M})d\mathbf{x}_v} \tag{7.5}$$

If we are given a nominally perfect map (i.e. told to take it as absolute truth) then $p(\mathbf{M}) = 1$ this simplifies the above:

$$p(\mathbf{x}_v|\mathbf{M},\mathbf{Z^k}) = \frac{p(\mathbf{Z^k}|\mathbf{M},\mathbf{x}_v)p(\mathbf{x}_v|\mathbf{M})p(\mathbf{M})}{\int_{-\infty}^{\infty} p(\mathbf{Z^k}|\mathbf{M},\mathbf{x})p(\mathbf{x}_v|\mathbf{M})p(\mathbf{M})d\mathbf{x}_v} \tag{7.6}$$

$$= \frac{p(\mathbf{Z^k}|\mathbf{x}_v)p(\mathbf{x}_v)}{\int_{-\infty}^{\infty} p(\mathbf{Z^k}|\mathbf{x})p(\mathbf{x}_v)d\mathbf{x}_v} \tag{7.7}$$

$$= \frac{p(\mathbf{Z^k}|\mathbf{x}_v)p(\mathbf{x}_v)}{\mathbf{C}(\mathbf{Z^k})} \tag{7.8}$$

7.4.2 Probabilistic Mapping

For the mapping side of things we are given the vehicle location (probably derived from absolute observation) and an sequence of vehicle relative observations. We wish to find the distribution of $\mathbf{M}$ conditioned on $\mathbf{Z^k}$and $\mathbf{x}_v$.

$$p(\mathbf{M}|\mathbf{x}_v,\mathbf{Z^k}) = \frac{p(\mathbf{Z^k}|\mathbf{M},\mathbf{x}_v)p(\mathbf{M},\mathbf{x}_v)}{p(\mathbf{x}_v,\mathbf{z})} \tag{7.9}$$

$$= \frac{p(\mathbf{Z^k}|\mathbf{M},\mathbf{x}_v)p(\mathbf{M}|\mathbf{x}_v)p(\mathbf{x}_v)}{p(\mathbf{x}_v,\mathbf{z})} \tag{7.10}$$

$$= \frac{p(\mathbf{Z^k}|\mathbf{M},\mathbf{x}_v)p(\mathbf{M}|\mathbf{x}_v)p(\mathbf{x}_v)}{\int_{-\infty}^{\infty} p(\mathbf{Z^k}|\mathbf{M},\mathbf{x}_v)p(\mathbf{M}|\mathbf{x}_v)p(\mathbf{x}_v)d\mathbf{M}} \tag{7.11}$$

$$\tag{7.12}$$

If we assume perfect knowledge of the vehicle location then $p(\mathbf{x}_v) = 1$ and so

$$p(\mathbf{M}|\mathbf{x}_v, \mathbf{Z^k}) = \frac{p(\mathbf{Z^k}|\mathbf{M}, \mathbf{x}_v)p(\mathbf{M}|\mathbf{x}_v)p(\mathbf{x}_v)}{\int_{-\infty}^{\infty} p(\mathbf{Z^k}|\mathbf{M}, \mathbf{x}_v)p(\mathbf{M}|\mathbf{x}_v)p(\mathbf{x}_v)d\mathbf{M}} \tag{7.13}$$

$$= \frac{p(\mathbf{Z^k}|\mathbf{M})p(\mathbf{M})}{\int_{-\infty}^{\infty} p(\mathbf{Z^k}|\mathbf{M})p(\mathbf{M})d\mathbf{M}} \tag{7.14}$$

$$= \frac{p(\mathbf{Z^k}|\mathbf{M})p(\mathbf{M})}{\mathbf{C}(\mathbf{Z^k})} \tag{7.15}$$

Equations 7.15 and 7.8 should have a familiar form — we met them when discussing Maximum a priori estimators, recursive bayesian estimation which collectively lead us to discuss and explore the Kalman filter. The Kalman filter would appear to be an excellent way in which to implement these equations. If we parameterise the random vectors $\mathbf{x}_v$ and $\mathbf{M}$ with first and second order statistics (mean and variance) then the Kalman Filter will calculate the MMSE estimate of the posterior. The first derivation of the Kalman filter presented proceeded by assuming Gaussian distributions. In this case the Kalman filter is the optimal Bayesian estimator. The Kalman filter provides a real time way to perform state estimation on board a vehicle.

7.5 Feature Based Estimation for Mapping and Localising

7.5.1 Feature Based Localisation

This is the simplest task. We are given a map $\mathbf{M}$ containing a set of features and a stream of observations of measurements between the vehicle and these features (see figure 7.3). We assume to begin with that an oracle is telling us the associations between measurements and observed features. We assume the vehicle we are navigating is equipped with a range-bearing sensor which returns the range and bearing to point like objects (the features). We will base the ensuing simulation on a real vehicle - the B21 we have already met - this means that we will have an additional sequence of dead-reckoned positions as input into the prediction stage. We denote these as $\mathbf{x}_o$

We have already have the prediction equations from previous discussions (Equation 7.16):

$$\hat{\mathbf{x}}(k|k-1) = \hat{\mathbf{x}}(k-1|k-1) \oplus (\ominus \mathbf{x}_o(k-1) \oplus \mathbf{x}_o(k)) \tag{7.16}$$
$$\hat{\mathbf{x}}(k|k-1) = \hat{\mathbf{x}}(k-1|k-1) \oplus \mathbf{u}_o(k) \tag{7.17}$$
$$\mathbf{P}(k|k-1) = \nabla\mathbf{F_x}\mathbf{P}(k-1|k-1)\nabla\mathbf{F_x}^T + \nabla\mathbf{F_v}\mathbf{Q}\nabla\mathbf{F_v}^T \tag{7.18}$$
$$= \mathbf{J}_1(\mathbf{x}_v, \mathbf{u}_o)\mathbf{P}_v(k-1|k-1)\mathbf{J}_1(\mathbf{x}_v, \mathbf{u}_o)^T + \mathbf{J}_2(\mathbf{x}_v, \mathbf{u}_o)\mathbf{U}_o\mathbf{J}_2(\mathbf{x}_v, \mathbf{u}_o)^T \tag{7.19}$$

Now we come to the observation equation which is simply a range r_i and bearing θ_i to the i^{th} feature:

$$\mathbf{z}(k) \triangleq \begin{bmatrix} r \\ \theta \end{bmatrix} \tag{7.20}$$
$$= \mathbf{h}(\mathbf{x}(k), \mathbf{w}(k)) \tag{7.21}$$
$$= \begin{bmatrix} \sqrt{(x_i - x_v(k))^2 + (y_i - y_v(k))^2} \\ atan2(\frac{y_i - y_v(k)}{x_i - x_v(k)}) - \theta_v \end{bmatrix} \tag{7.22}$$

We differentiate w.r.t $\mathbf{x}_v$ to arrive at the observation model jacobian:

$$\nabla\mathbf{H_x} \triangleq \frac{\partial \mathbf{h}}{\partial \mathbf{x}} \tag{7.23}$$
$$= \begin{bmatrix} \frac{x_i - x_v(k)}{r} & \frac{y_i - y_v(k)}{r} & 0 \\ \frac{y_i - y_v(k)}{r^2} & -\frac{x_i - x_v(k)}{r^2} & -1 \end{bmatrix} \tag{7.24}$$

We assume independent errors on range and bearing measurements and use a diagonal observation covariance matrix $\mathbf{R}$:

$$\mathbf{R} = \begin{bmatrix} \sigma_r^2 & 0 \\ 0 & \sigma_\theta^2 \end{bmatrix} \tag{7.25}$$

Section 11.3 is a print out of an implementation of feature based localisation using the models we have just discussed. You can also download the code from the course web-site *¡http://www.robots.ox.ac.uk/∼pnewman : teaching¿*. Figure 7.5.1 shows the trajectory of the vehicle as it moves through a field of random point features. Note that the code simulates a sensor failure for the middle twenty percent of the mission. During this time the vehicle becomes more and more lost. When the sensor comes back on line there is a jump in estimated vehicle location back to one close to the true position (see figure 7.5.1).

7.5.2 Feature Based Mapping

Now we will consider the dual of Localisation - Mapping. In this case the vehicle knows where it is but not what is in the environment. Perhaps the vehicle is fitted with a GPS

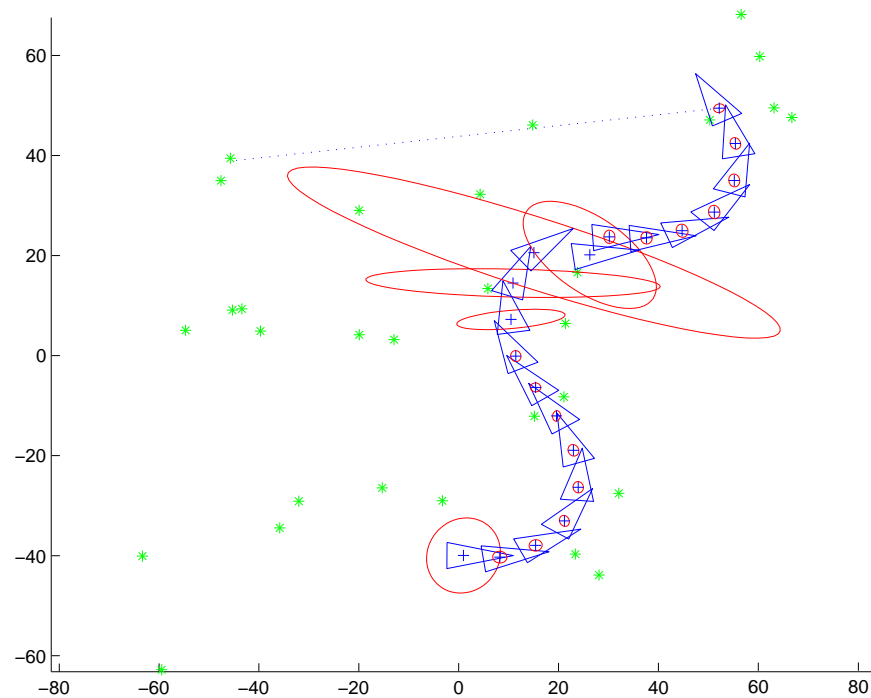



Figure 7.2: Feature Based Localisation

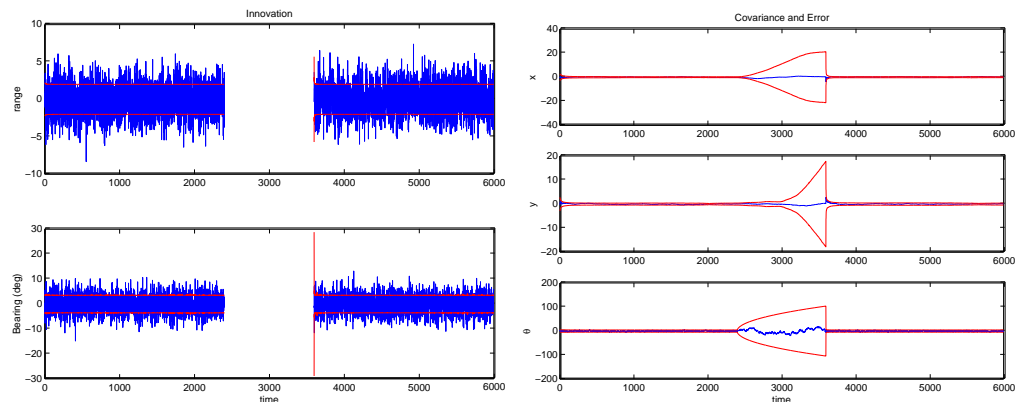



Figure 7.3: Feature Based Localisation innovation and error/covariance plots. Notice how when no measurements are made the vehicle uncertainty grows rapidly but reduces when features are re-observed. The covariance bounds are 3-sigma and 1-sigma-bound on state error and innovation plots respectively.

or some other localisation system using an a-priori map. To avoid initial confusion we'll imagine the vehicle has a 'super-gps' on board telling it where it is at all times.

The state vector for this problem is now much larger -it will be the Map itself and

the concatenation of all point features. The observation equation is the same as for the localisation case only now the feature co-ordinates are the free variables.

The prediction model for the state is trivial. We assume that features don't move and so $\mathbf{x}(k+1|k)_{map} = \mathbf{x}(k|k)_{map}$.

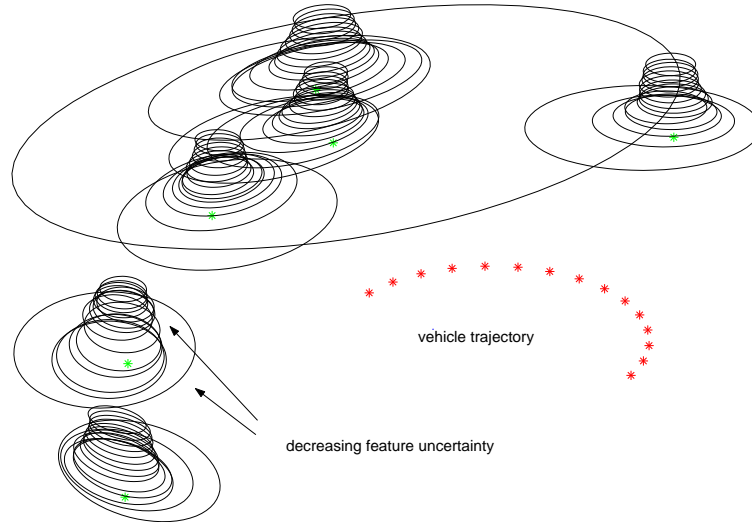



Figure 7.4: Evolution of a map over time. The vertical axis is time (k). Covariance ellipses for each feature are plotted in the $z = k$ planes. Note how the filter converges to a perfect map because no process noise is added in the prediction step

Initially the map is empty and so we need some method to add "newly discovered' features to it. To this end we introduce a feature initialisation function $\mathbf{Y}$ that take as arguments the old state vector and an observation to a landmark and returns a new, longer state vector with the new feature at its end. For the case of a range bearing measurement we would have the following:

$$\mathbf{x}(k|k)^* = \mathbf{y}(\mathbf{x}(k|k), \mathbf{z}(k), \mathbf{x}_v(k|k)) \tag{7.26}$$

$$= \begin{bmatrix} \mathbf{x}(k|k) \\ \mathbf{g}(\mathbf{x}(k), \mathbf{z}(k), \mathbf{x}_v(k|k)) \end{bmatrix} \tag{7.27}$$

$$= \begin{bmatrix} \mathbf{x}(k|k) \\ x_v + r\cos(\theta + \theta_v) \\ y_v + r\sin(\theta + \theta_v) \end{bmatrix} \tag{7.28}$$

$$\tag{7.29}$$

where the coordinates of the new feature are given by the function $\mathbf{g}$:

$$\mathbf{x}_{fnew} = \mathbf{g}(\mathbf{x}(k), \mathbf{z}(k), \mathbf{x}_v(k|k)) \tag{7.30}$$

$$= (\begin{bmatrix} x_v + r\cos(\theta + \theta_v) \\ y_v + r\sin(\theta + \theta_v) \end{bmatrix} \tag{7.31}$$

We also need to figure out how to transform the covariance matrix $\mathbf{P}$ when adding a new feature. Of course we can use the jacobian of the transformation:

$$\mathbf{P}(k|k)^* = \nabla\mathbf{Y}_{\mathbf{x},\mathbf{z}} \begin{bmatrix} \mathbf{P}(k|k) & 0 \\ 0 & \mathbf{R} \end{bmatrix} \nabla\mathbf{Y}_{\mathbf{x},\mathbf{z}}{}^T \tag{7.32}$$

where

$$\nabla\mathbf{Y}_{\mathbf{x},\mathbf{z}} = \begin{bmatrix} \mathbf{I}_{n\times n} & \mathbf{0}_{\mathbf{n}\times\mathbf{2}} \\ \nabla\mathbf{G}_{\mathbf{x}} & \nabla\mathbf{G}_{\mathbf{z}} \end{bmatrix} \tag{7.33}$$

Now for the mapping case $\nabla\mathbf{G}_{\mathbf{x}} = 0$ as the new feature is not dependent on any element in the state vector (which only contains features). Therefore:

$$\mathbf{P}(k|k)^* = \begin{bmatrix} \mathbf{P}(k|k) & 0 \\ 0 & \nabla\mathbf{G}_{\mathbf{z}}\mathbf{R}\nabla\mathbf{G}_{\mathbf{z}}{}^T \end{bmatrix} \tag{7.34}$$

One thing to realise is that the observation Jacobian is now "long and thin". When observing feature i it is only non-zero at the "location" (indexes) of the feature in the state vector:

$$\nabla\mathbf{H}_{\mathbf{x}} = \Big[\underbrace{\cdots\mathbf{0}\cdots}_{other features} \underbrace{\nabla\mathbf{H}_{\mathbf{x_{fi}}}}_{observedfeature} \underbrace{\cdots\mathbf{0}\cdots}_{other features} \Big] \tag{7.35}$$

Figure 7.5.2 shows the evolution of the map over time. Note that as no process noise is ever added to the system the uncertainty in feature locations after initialisation is always decreasing. In the limit the map will be known perfectly. The code used to generate this simulation for feature based mapping can be downloaded from the course web-site. You might think it odd that in the limit the map becomes perfectly known. You might think that intuitively there should always be some residual uncertainty. The point is that the

vehicle is continually being told where (via the super-GPS) it is and is not changing its position estimate as a function of landmark observations. As a consequence all features are independent (check this by examining the **P** matrix –it is block diagonal) and each observation of them simply adds information and therefore reduces their uncertainty - again and again and again.... This is in contrast to the next section where landmark observations will be allowed to adjust map and vehicle estimates simultaneously!

7.6 Simultaneous Localisation and Mapping - SLAM

SLAM is the generalised navigation problem. It asks if it is possible for a robot, starting with no prior information, to move through its environment and build a consistent map of the entire environment. Additionally the vehicle must be able to use the map to navigate (localise) and hence plan and control its trajectory during the mapping process. The applications of a robot capable of navigating , with no prior map, are diverse indeed. Domains in which 'man in the loop' systems are impractical or difficult such as Martian exploration, sub-sea surveys, and disaster zones are obvious candidates. Beyond these, the sheer increase in autonomy that would result from reliable, robust navigation in large dynamic environments is simply enormous. Autonomous navigation has been an active area of research for many years.

In SLAM no use is made of prior maps or external infrastructure such as GPS. A SLAM algorithm builds a consistent estimate of both environment and vehicle trajectory using only noisy proprioceptive (e.g., inertial, odometric) and vehicle-centric (e.g., radar, camera and laser) sensors. Importantly, even though the relative observations are of the local environment, they are fused to create an estimate of the *complete* workspace.

The SLAM problem is of fundamental importance in the quest for autonomous mobile machines. It binds aspects of machine learning, uncertainty management, perception, sensing and control to enable a machine to discover and understand its surroundings with no prior knowledge or external assistance.

This section will introduce a simple feature based approach to the SLAM problem. It doesn't work in real life for deployments in large areas because it involves running a Kalman filter to estimate the entire map and vehicle state. The update of the covariance matrix is therefore at best proportional to the square of the number of features. Given enough features the system will grind to a halt. Figure 7.6 shows some of the kind of area that can be mapped and localised in using this technique.

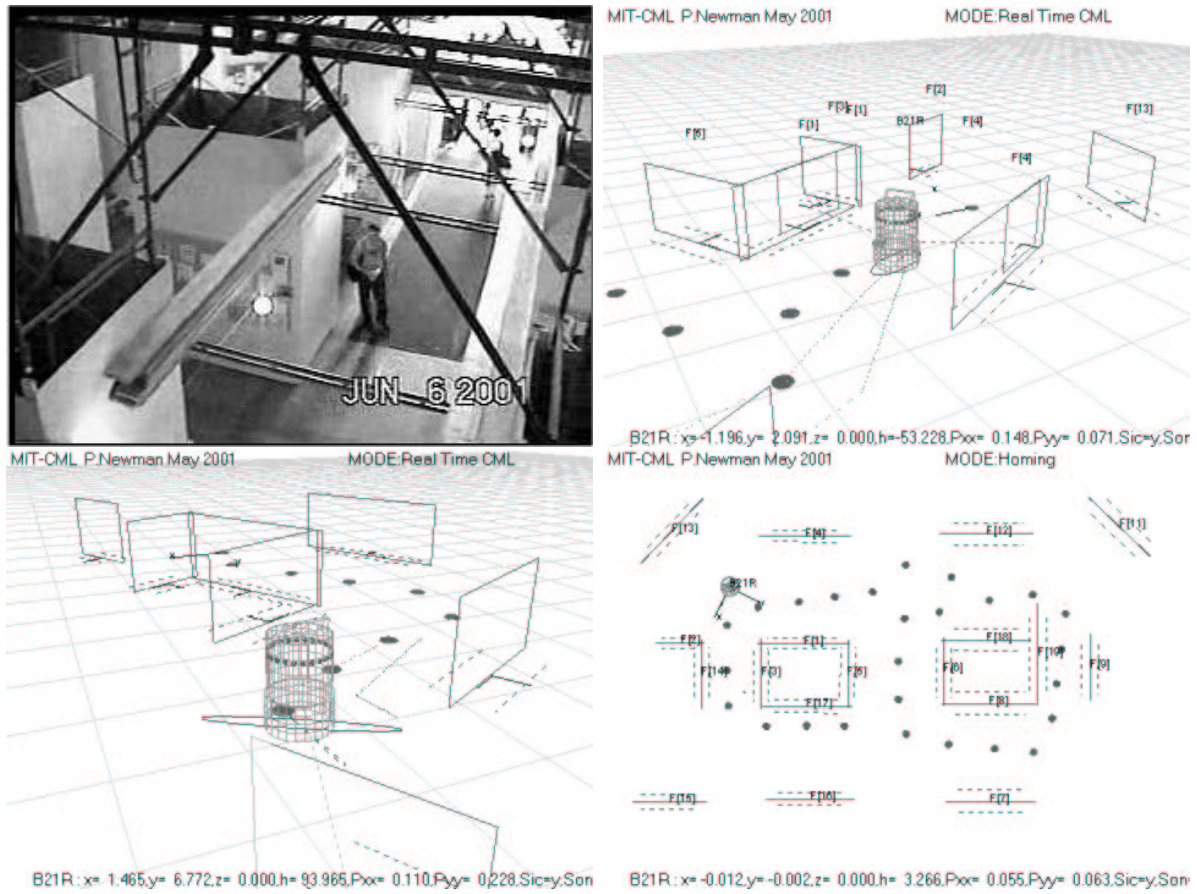



Figure 7.5: SLAM in a real environment. These figures are screen shots from an experiment using a B21 robot in a hallway at MIT. Having built the map the vehicle used it to navigate home to within a few cm of its start position. Videos can be found at *¡http://www.robots.ox.ac.uk/∼pnewman : teaching¿*.

You'll be pleased to know we have pretty much done all the work required to implement a full SLAM algorithm. We will still employ the oracle that tells us which feature is being seen with each observation. All we do now is change our state vector to include *both* vehicle and map:

$$\mathbf{x} \triangleq \begin{bmatrix} \mathbf{x}_v \\ \mathbf{x}_{\mathbf{f},1} \\ \vdots \\ \mathbf{x}_{\mathbf{f},n} \end{bmatrix} \tag{7.36}$$

Of course to start with $n = 0$ but as new features are seen the state vector is grown

just as in the pure mapping case. The difference now is that the observation and feature initialisation jacobians have two non-zero blocks. one with respect to the vehicle and one with respect to the observed feature.

$$\nabla \mathbf{H_x} = \left[\nabla \mathbf{H_{x_v}} \cdots \nabla \mathbf{H_{x_{f,1}}} \right] \tag{7.37}$$

$$\nabla \mathbf{Y_{x,z}} = \begin{bmatrix} \mathbf{I}_{n\times n} & \mathbf{0_{n\times 2}} \\ \nabla \mathbf{G_x} & \nabla \mathbf{G_z} \end{bmatrix} \tag{7.38}$$

$$= \begin{bmatrix} \mathbf{I}_{n\times n} & \mathbf{0_{n\times 2}} \\ \left[\nabla \mathbf{G_{x_v}} \cdots \mathbf{0} \cdots \right] & \nabla \mathbf{G_z} \end{bmatrix} \tag{7.39}$$

Also note that the jacobian of the prediction models have changed in an obvious way:

$$\nabla \mathbf{F_x} = \begin{bmatrix} \nabla \mathbf{F_{x_v}} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{2n\times 2n} \end{bmatrix} \tag{7.40}$$

$$= \begin{bmatrix} \mathbf{J}1(\mathbf{x}_v, \mathbf{u}) & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{2n\times 2n} \end{bmatrix} \tag{7.41}$$

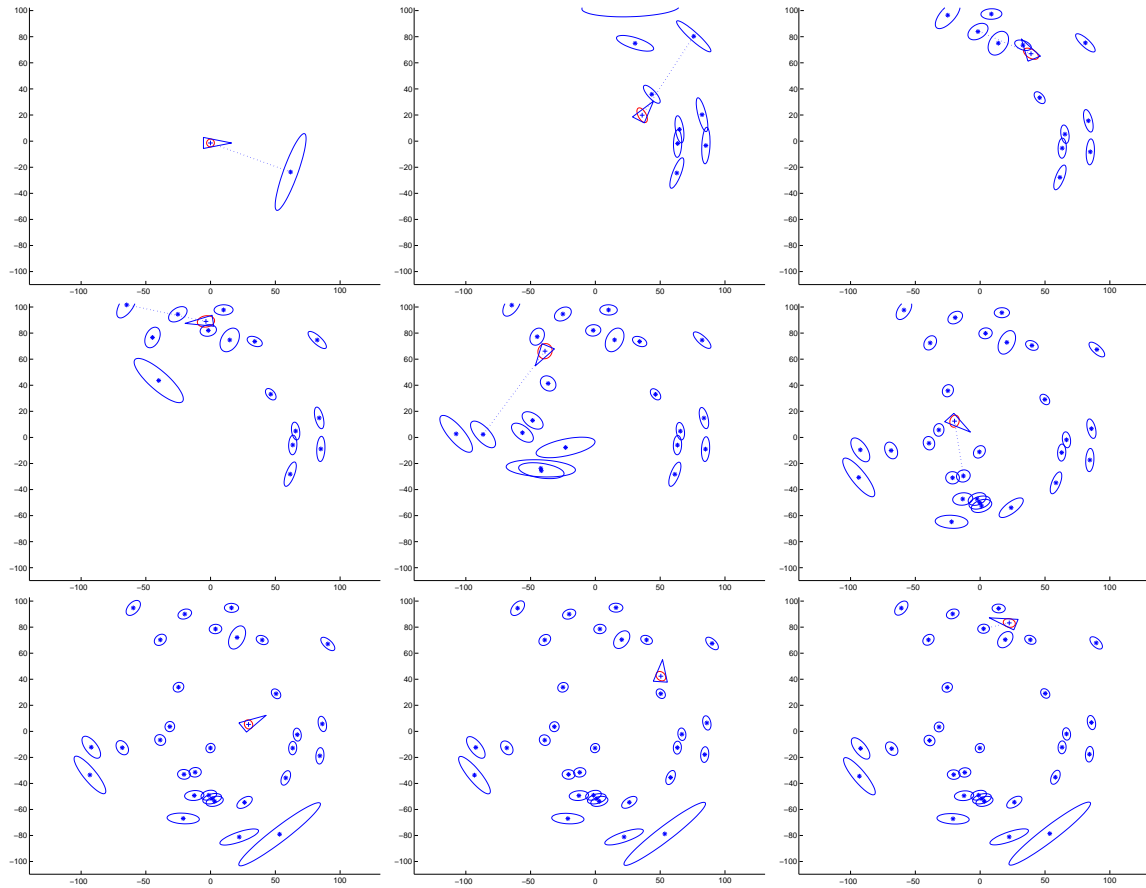
$$\nabla \mathbf{F_u} = \begin{bmatrix} \mathbf{J}2(\mathbf{x}_v, \mathbf{u}) & \mathbf{0} \\ \mathbf{0} & \mathbf{0_{2n\times 2n}} \end{bmatrix} \tag{7.42}$$

$$\tag{7.43}$$

because the prediction model is now:

$$\begin{bmatrix} \mathbf{x}_v(k+1) \\ \mathbf{x}_{\mathbf{f},1}(k) \\ \vdots \\ \mathbf{x}_{\mathbf{f},n}(k) \end{bmatrix} = \begin{bmatrix} \mathbf{x}_v(k) \oplus \mathbf{u}(k) \\ \mathbf{x}_{\mathbf{f},1}(k) \\ \vdots \\ \mathbf{x}_{\mathbf{f},n}(k) \end{bmatrix} \tag{7.44}$$

Note that the features are not expected to move between time steps (pretty much what you would hope for when using a map!) and they are noiseless. Only the vehicle has process noise injected into its covariance matrix during a prediction step.

The matlab code for an EKF, feature based SLAM algorithm can be found on the course web-site. You should be able to recognise it as a union of previous localisation and mapping examples. Figure 7.6 shows a few snap shots of the algorithm running.

7.6.1 The role of Correlations

Note that the covariance matrix now has some structure to it -you can partition map $\mathbf{P}_{mm}$ and vehicle $\mathbf{P}_{vv}$ blocks.

$$\mathbf{P} = \begin{bmatrix} \mathbf{P}_{vv} & \mathbf{P}_{vm} \\ \mathbf{P}_{vm}^T & \mathbf{P}_{mm} \end{bmatrix} \tag{7.45}$$

Of diagonals $\mathbf{P}_{vm}$ are the correlations between map and vehicle. Its pretty clear that there should be correlations between map and vehicle as they are so interrelated. Consider the following sequence of events. From the moment of initialisation the feature's location is a function of vehicle location and so errors in the vehicle location will also appear as errors in feature location. Now recall the discussion regarding correlations in section 5.1.3 — correlations cause adjustments in one state to ripple into adjustments in other states. This is of course also true in this kind of approach to the SLAM problem. Remarkably every observation of a feature affects the estimate of every other feature in the map. It's as though they are all tied up together with elastic bands - pulling at one will pull at the others in turn.

There are some further characteristics of the SLAM problem that transcend the estimation method being used. You should be able to check that they are true by running the example code:

- The feature uncertainties never drop below the initial uncertainty of the vehicle. This makes sense, if you start with say 10m of uncertainty relative to Carfax tower mapping and re-observing things within the Thom building is never going to reduce your overall uncertainty. The best you can hope for is to reduce all features to the lower bound - your initial uncertainty. Compare this to the ever decreasing uncertainty of features in the mapping only case.
- The feature uncertainties never increase but the vehicle uncertainty can. The prediction model is the identity matrix for the map - we don't expect it to move. Furthermore the map has a noiseless model and so the prediction step does not inflate the covariance of the map.

The SLAM problem is a very topical problem and is attracting interest from the AI, robotics and machine learning communities. If you want to find out more just ask....

Topic 8

Multi-modal and other Methods

The previous discussion on Estimation, Mapping and Localisation has focused almost exclusively on estimating uni-model pdfs. Much use has been made of the Kalman filter which estimates the mean and covariance of a pdf which under gaussian noise assumptions is the MMSE estimator. The Kalman filter is an important tool (and has great value outside of this course) but it is not the whole story.

The single EKF approach we have talked about so far does not really behave well in large environments - the big problem is that linearisation errors (that came from the Taylor series approximations in deriving the EKF) compound and the linearisation starts to occur around points that are far from the true state. There is neither the space nor time to cover in detail how this can be overcome. Broadly though, one of two schemes are adopted. The first partitions the world into local maps and uses uni-modal methods (EKF) within each map. The maps are then carefully glued back together to form a unified global map. Secondly uni-modal methods are abandoned altogether. Instead, monte-carlo methods are used.

8.1 Montecarlo Methods - Particle Filters

The basic idea is simple. We maintain lots of different versions of the state vector — all slightly different from each other. When a measurement comes in we score how well each version explains the data. We then make copies of the best fitting vectors and randomly perturb them (the equivalent of adding $\mathbf{Q}$ in the EKF) to form a new generation of candidate states. Collectively these thousands of possible states and their scores define the pdf we are seeking to estimate. We never have to make the assumption of Gaussian noise or perform

a linearisation. The big problem with this elegant approach is the number of sample-states (versions) we need to try out increase geometrically with the number of states being estimated. For example to estimate a 1D vector we may keep 100 samples. For a 2D vector we would require something like 100^2, for 3D 100^3 and so on. The trick often played is to be smart about selecting which samples should be copied between timesteps. This is a very current area of research.

The PF-Localisation algorithm proceeds as follows:

1. Initialise n particles in a map. Each particle is a 3 by 1 state vector of the vehicle
2. Apply the plant model to each particle. In contrast to the prediction step in the Kalman approach we actually inject the process noise as well — i.e we add a random vector to the control vector $\mathbf{u}$. This is in some ways analogous to adding $\mathbf{GUG}^T$ to the predicted covariance in the Kalman approach because it increases the variance of the estimate (in this case the particle set).
3. For each particle predict the observation. Compare this to the measured value. Use a likelihood function to evaluate the likelihood of the observation given the state represented by each particle. This will be a scalar L which is associated with each particle. This scalar is referred to as a “weight” or “importance”.
4. Select the particles that best explain the observation. One way to do this would be to simply sort and select the top candidates based on L but this would reduce the number of particles. One solution to this would be to copy from this “winning set” until we have n particles again. This has a hidden consequence though: it would artificially reduce the diversity (spread) of the particle set. Instead we randomly sample from the samples biased by the importance values. This means that there is a finite chance that particles that really didn’t explain the observation well, will be reselected. This may turn out to be a good plan because they may do a better job for the next observation.
5. Goto 2

Figure 8.1 shows this process graphically for the localisation problem. A common question is “what is the estimate?”. Well technically it is the distribution of particles themselves which represents a pdf. If you want a single vector estimate, the mode, mean and median are all viable options. However you need to think carefully. If you run the code yourself initially you will see distinct clouds of particles representing differing modes of the pdf - the mean is somewhere between them all in a location with no support[1]! As with all the algorithms

[1]however soon the clouds should coverge into one large cloud as the motion of the vehicle disambiguates its location

in the course you are strongly advised to down load the code (MCL.m) and peruse it to understand the translation from concept to implementation. In summary, the particle filter approach is very elegant and very simple to implement (see the code on the website). A crucial point is that it does not require any linearisation assumptions (there are no jacobians involved) and there are no Gaussian assumptions. It is particularly well suited for problems with small state spaces - the example code has a state dimension of three.

8.2 Grid Based Mapping

Up until now we have adopted a feature based approach to navigation. We assumed that the environment contains features possessing some geometry that can be extracted by suitable processing of sensor data - edges, points, lines etc. We then store and estimate (using further measurements) this sparse parameterisation (often by stacking all the feature parameters in to a long state vector and calling it $\mathbf{x}$).

In contrast grid based approaches tend not to reason about the geometry of features in the environment but take a more sensor-centric view. A mesh of cells (a grid) is laid over the (initially unknown) world. Each cell $c(i,j)$ has a number associated with it corresponding to an occupancy probability - $\mathbf{p}_o(i,j)$ and a probability of being empty $\mathbf{p}_e$. So for example if $\mathbf{p}_o(40,2) = 1$ we are sure that the cell at (40,20) is occupied. We make no assertions as to what is occupying it just that something or part of something is at the coordinates covered by cell(40,20).

We use two functions to model the behaviour of the sensor - one to express the likelihood of a cells within the observed region being empty and one to express the likelihood of them being occupied. Figure 8.2 shows one possible set of functions for a widebeam in-air sonar. For each cell under the beam a Bayes rule update is used to figure out the new probability of a cell being empty or being occupied. As the sensor (mounted on a vehicle) moves around the environment and **if** the location of the sensor is known a grid based map can be built. Figure 8.2 shows the occupancy and empty maps for a map built with the CMU Terragator vehicle in figure 8.2.

One problem with grid based methods is memory usage - it becomes expensive for 3D maps. However cunning data structures can overcome this. A second issue is that of grid alignment - it is arbitrary and as such cells are likely to cover areas that are awkwardly partially full as such is not possible to capture the sharp discontinuities at the edges of objects. Perhaps the biggest problem of grid based methods stems from the difficulties in localising with respect to a grid. It requires trying to match a small sensor centric map with

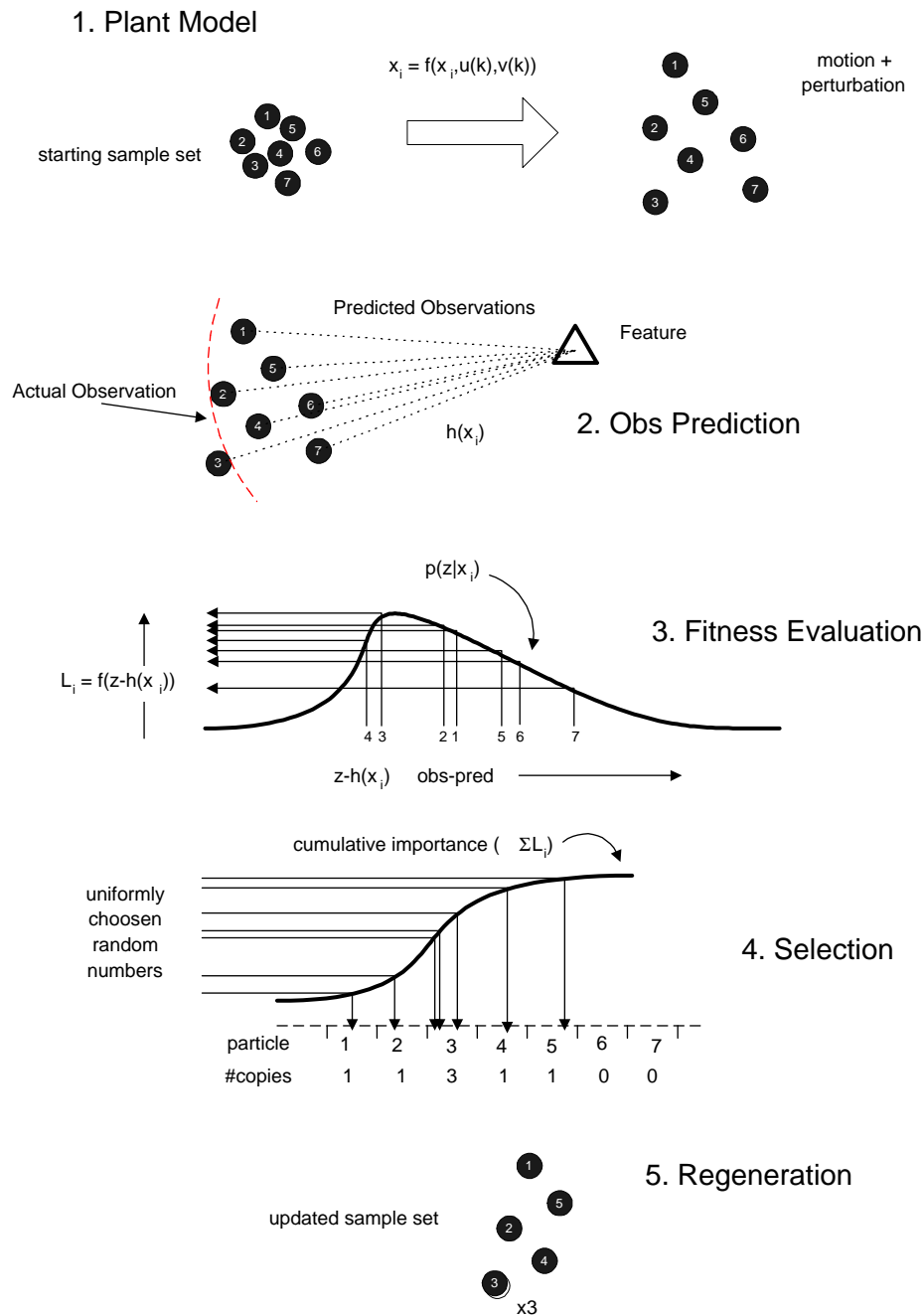



Figure 8.1: Graphical interpretation of a particle filter. The method is general but this diagram can be easily understood in terms of a range-to-beacon sensor (like the long baseline acoustic sensor mentioned in the Least Squares section). The best particles are those that best explain the actual range measurement. These are preferentially choosen to form the next generation.

Fig. 3. Terregator outdoors robot.

Figure 8.2: From the early days of mobile robotics. The CMU terragator vehicle used grid based mapping in various settings.

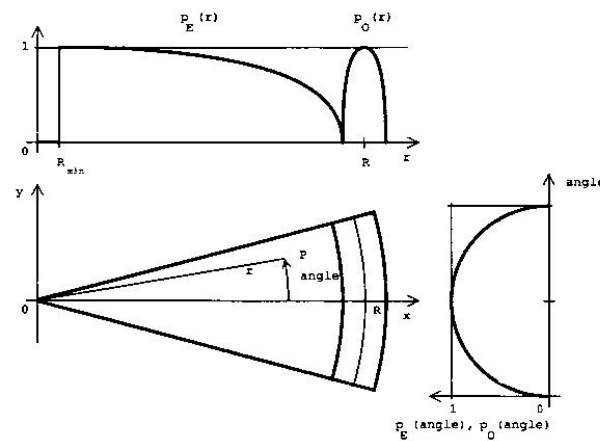



Figure 8.3: The shape of two functions used to describe the probability $P_e(r)$ of empty cells at a range r from the sensor and the probability of them being occupied $p_o(r)$. The actual object is at distance R from the sensor. The sensor modelled here is a wide beam sonar that sends out a cone of sound and measures the time between transmit and echo reception.

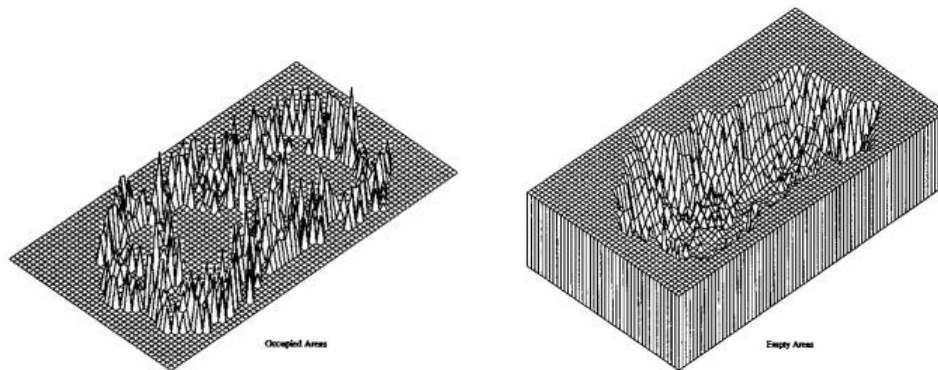



Figure 8.4: The two typical occupancy maps built by a grid based approach. The left map show the probability of regions being occupied and the left the probability of regions being empty. Each map is maintained using a different likelihood function

a large global map. The vehicle can be localised by rotating and translating the local map (what is seen by the sensor) until the best match is found.

Topic 9

In Conclusion

Hopefully you have found this introductory course interesting and have picked up a sense that the tools we have been using are applicable to many domains. If this were a 10 lecture course we would have spent more time considering how to appraise in real-time the validity of sensor data. This is a crucial point — we need to decide whether the data is even worth processing. In may situations sensors return grossly erroneous data (outliers). However if we don't know what the world looks like how can we be sure it is bad data and not faithful sensing of an unexpected world state? I also hope you have found the discussions on mobile robotics thought provoking. Perhaps in a decade we will have cars that avoid crashes, goods trucks that drive themselves, an always-deployed fleet of ocean vehicles monitoring El-Nino currents, smart machines on Mars and robots that massively increase the independence and mobility of our old and infirm. We don't need the "full-monty" of AI to be able to make a big difference....(but it would be useful).

P.M.N October 2003.

Topic 10

Miscellaneous Matters

This section is meant to help you with the class work and covers things that you may need to know but that don't really fit in the main body of the notes

10.1 Drawing Covariance Ellipses

The n-dimensional multivariate-normal distribution has form

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{n/2} \mid \mathbf{P} \mid^{1/2}} \exp\{-\frac{1}{2}(\mathbf{x} - \mu_x)^T \mathbf{P}^{-1}(\mathbf{x} - \mu_x)\} \tag{10.1}$$

The exponent $(\mathbf{x} - \mu_x)^T \mathbf{P}^{-1}(\mathbf{x} - \mu_x)$ defines contours on the bell curve. A particularly useful contour is the "$1 - \sigma$ bound" where $(\mathbf{x} - \mu_x)^T \mathbf{P}^{-1}(\mathbf{x} - \mu_x) = 1$. It is really useful to be able to plot this contour to illustrate uncertainties in 2D. We can disregard the vector μ_x as it simply shifts the distribution. So we have

$$\mathbf{x}^T \mathbf{P}^{-1} \mathbf{x} = 1 \tag{10.2}$$

we can use eigenvalue decomposition to factorise $\mathbf{P}$ as $\mathbf{V}\mathbf{D}\mathbf{V}^T$ where $\mathbf{V}$ is some rotation matrix and $\mathbf{D}$ is a diagonal matrix of principal values

$$\mathbf{x}^T (\mathbf{V}\mathbf{D}\mathbf{V}^T)^{-1} \mathbf{x} = 1 \tag{10.3}$$

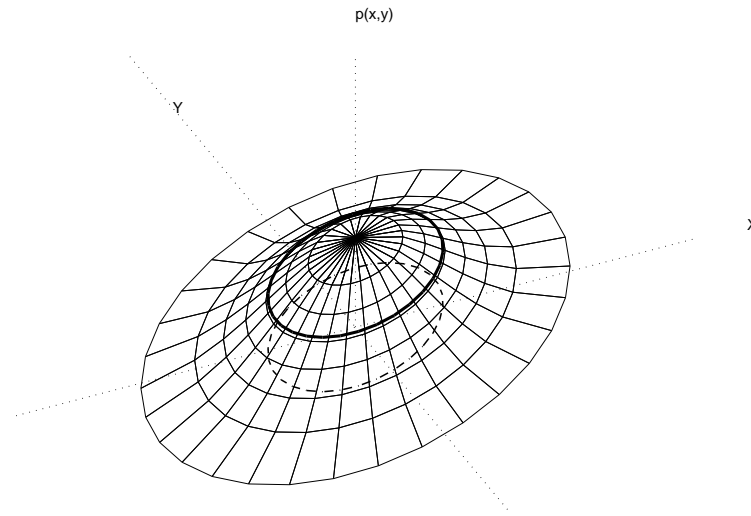



Figure 10.1: A bi-variate normal distribution. The thick contour is the $1-\sigma$ bound where $(\mathbf{x}-\mu_x)^T\mathbf{P}^{-1}(\mathbf{x}-\mu_x) = 1$

.

using the fact that $\mathbf{V}$ is orthonormal:

$$\mathbf{x}^T\mathbf{V}\mathbf{D}^{-1}\mathbf{V}^T\mathbf{x} = 1 \tag{10.4}$$

$$\mathbf{x}^T\mathbf{V}\mathbf{D}^{-1/2}\mathbf{D}^{-1/2}\mathbf{V}^T\mathbf{x} = 1 \tag{10.5}$$

$$\mathbf{x}^T\mathbf{K}\mathbf{K}^T\mathbf{x} = 1 \tag{10.6}$$

where

$$\mathbf{K} = \mathbf{V}\mathbf{D}^{-1/2}. \tag{10.7}$$

now for any point $\mathbf{y} = [x, y]^T$ which is on the unit circle, $\mathbf{y}^T\mathbf{y} = 1$, so

$$\mathbf{x}^T\mathbf{K}\mathbf{K}^T\mathbf{x} = \mathbf{y}^T\mathbf{y} \tag{10.8}$$

$$\mathbf{K}^T\mathbf{x} = \mathbf{y} \tag{10.9}$$

$$\Rightarrow \mathbf{x} = \mathbf{V}\mathbf{D}^{1/2}\mathbf{y} \tag{10.10}$$

So to draw an ellipse described by a 2×2 covariance matrix $\mathbf{P}$ we take a whole bunch of points on the unit circle and multiply them by $\mathbf{V}\mathbf{D}^{1/2}$ and plot the resulting points. This makes sense, intuitively the variance is in "units squared"— $\mathbf{D}^{1/2}$ is a diagonal matrix of standard deviations - the semi-major and semi-minor axes of the ellipse. The fist thing we do is scale the unit circle by these factors. Then we rotate the ellipse by $\mathbf{V}$ — recall that the correlations between x and y (off diagonals in $\mathbf{P}$) induce a rotation in the ellipse?

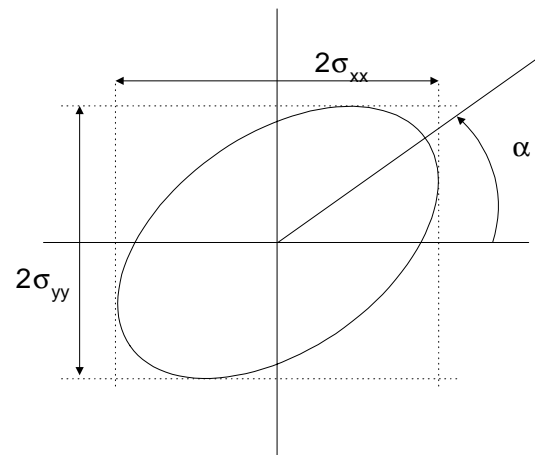



Figure 10.2: The relationship between geometry of the $1 - \sigma$ bound for a bivariate normal distribution and its covariance matrix.

Figure 10.1 shows the relation ship between $\mathbf{P}$ and the plotted form for a bivariate normal distribution.

$$\mathbf{P} = \begin{bmatrix} \sigma_{xx}^2 & \sigma_{xy}^2 \\ \sigma_{xy}^2 & \sigma_{yy}^2 \end{bmatrix} \tag{10.11}$$

$$\tan 2\alpha = \frac{2\sigma_{xy}^2}{\sigma_{xx}^2 - \sigma_{yy}^2} \tag{10.12}$$

10.2 Drawing High Dimensional Gaussians

Obviously it is hard to draw a high dimensional gaussian on a screen or paper. For example we may have a covariance matrix that corresponds to a large state vector:

$$\mathbf{x} = \begin{bmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \end{bmatrix} \tag{10.13}$$

$$\mathbf{P} = \begin{bmatrix} \left.\begin{bmatrix} {\sigma_{a_{11}}}^2 & {\sigma_{a_{12}}}^2 \\ {\sigma_{a_{21}}}^2 & {\sigma_{a_{22}}}^2 \end{bmatrix}\right\} \mathbf{P}_{aa} & \cdots & \cdots \\ \vdots & \overbrace{\left.\begin{bmatrix} {\sigma_{b_{11}}}^2 & {\sigma_{b_{12}}}^2 \\ {\sigma_{b_{21}}}^2 & {\sigma_{b_{22}}}^2 \end{bmatrix}\right\} \mathbf{P}_{bb}}^{\text{marginal of component } \mathbf{b}} & \cdots \\ \vdots & \vdots & \left.\begin{bmatrix} {\sigma_{c_{11}}}^2 & {\sigma_{c_{12}}}^2 \\ {\sigma_{c_{21}}}^2 & {\sigma_{c_{22}}}^2 \end{bmatrix}\right\} \mathbf{P}_{cc} \end{bmatrix} \tag{10.14}$$

So to plot a 2D representation of the i^{th} entity in a state vector simply plot the ellipse for the i^{th} 2 by 2 block in $\mathbf{P}$.

Topic 11

Example Code

11.1 Matlab Code For Mars Lander Example

```
%----------------------------------------------------------
% A MARS LANDER --- EXAMPLE CODE C4
% P. Newman 2003
%----------------------------------------------------------


%--------- TOP LEVEL LOOP -----------------------%
function SimulateMarsLander
close all;clear all;

global Params;global XTrue;global VehicleStatus;global Store;

%set up parameters
 DoInitialise;

%initial conditions of estimator
XEst = [Params.X0+Params.InitialHeightError;Params.V0+Params.InitialVelocityError];
PEst = diag([Params.InitialHeightStd^2, Params.InitialVelocityStd^2]);

%store initial conditions:
DoStore(1,XEst,PEst,[0],[0],NaN);

k = 2;
while(~VehicleStatus.Landed & k <Params.StopTime/Params.dT)

    % simulate the world
    DoWorldSimulation(k);
```

```
    % read from sensor
    z(k) = GetSensorData(k);

    % estimate the state of the vehicle
    [XEst,PEst,S,Innovation] = DoEstimation(XEst,PEst,z(k));

    % make decisions based on our esimated state
    DoControl(k,XEst);

    % store results
    DoStore(k,XEst,PEst,Innovation,S,z(k));

    %tick...
    k = k+1;
end;

%draw pictures....
DoMarsGraphics(k);

return;

%--------- PROBLEM SET UP AND INITIALISATION ----------------------%
% users changes parameters here
function DoInitialise
global Params;
global XTrue;
global VehicleStatus;
global Store;

%------ user configurable parameters -------------
Params.StopTime = 600;%run for how many seconds (maximum)?
Params.dT = 0.1; % Run navigation at 10Hz
Params.c_light = 2.998e8;
Params.EntryDrag = 5;                           % linear drag constant
Params.ChuteDrag = 2.5*Params.EntryDrag;  % linear drag constant with chute open
Params.g = 9.8/3;  % assumed gravity on mars
Params.m = 50;     % mass of vehcile
Params.RocketBurnHeight = 1000; % when to turn on brakes
Params.OpenChuteHeight = 4000; %when to open chute
Params.X0 = 10000; % true entry height
Params.V0 = 0;     % true entry velocity
Params.InitialHeightError = 0; % error on entry height
Params.InitialVelocityError = 0; % error on entry velocity
Params.InitialHeightStd = 100; %uncertainty in initial conditions
Params.InitialVelocityStd = 20; %uncertainty in initial conditions
Params.BurnTime = NaN;
Params.ChuteTime = NaN;
Params.LandingTime = NaN;

%initial vehicle condition at entry into atmosphere...
VehicleStatus.ChuteOpen = 0;
VehicleStatus.RocketsOn = 0;
VehicleStatus.Drag = Params.EntryDrag;
VehicleStatus.Thrust = 0;
```

```
VehicleStatus.Landed = 0;

%process plant model (constant velcoity with noise in acceleration)
Params.F = [1 Params.dT;
            0 1];

%process noise model (maps acceleration noise to other states)
Params.G = [Params.dT^2/2 ;Params.dT];

%actual process noise truely occuring - atmospher entry is a bumpy business
%note this noise strength - not the deacceleration of the vehicle ....
Params.SigmaQ = 0.2; %ms^{-2}

%process noise strength how much acceleration (varinace) in one tick
% we expect (used to 'explain' inaccuracies in our model)
%the 3 is scale factor (set it to 1 and real and modelled noises will
%be equal
Params.Q = (1.1*Params.SigmaQ)^2; %(ms^2 std)

%observation model (explains observations in terms of state to be estimated)
Params.H = [2/Params.c_light 0];

%observation noise strength (RTrue) is how noisey the sensor really is
Params.SigmaR = 1.3e-7; %(seconds) 3.0e-7 corresponds to around 50m error....

%observation expected noise strength (we never know this parameter exactly)
%set the scale factor to 1 to make model and reallity match
Params.R = (1.1*Params.SigmaR)^2;

%initial conditions of (true) world:
XTrue(:,1) = [Params.X0;Params.V0];

Params
return;

%------------------- MEASUREMENT SYSTEM -------------------%
function z = GetSensorData(k)
global XTrue;
global Params;
    z = Params.H*XTrue(:,k) + Params.SigmaR* randn(1);
return;

%---------------- ESTIMATION KALMAN FILTER ----------------%
function [XEst,PEst,S,Innovation] = DoEstimation(XEst,PEst,z)
global Params;
F = Params.F;G = Params.G;Q = Params.Q;R = Params.R;H = Params.H;

%prediction...
XPred = F*XEst;
PPred = F*PEst*F'+G*Q*G';

% prepare for update...
Innovation = z-H*XPred;
S = H*PPred*H'+R;
W = PPred*H'*inv(S);
```

```
% do update....
XEst = XPred+W*Innovation;
PEst = PPred-W*S*W';
return;

%---------------- ITERATE SIMULATION -----=----------------%
function DoWorldSimulation(k)

global XTrue;global Params;global VehicleStatus;

oldvel = XTrue(2,k-1);
oldpos = XTrue(1,k-1);
dT = Params.dT;

%friction is a function of height
cxtau = 500; % spatial exponential factor for atmosphere density)
AtmospherDensityScaleFactor = (1-exp(-(Params.X0-oldpos)/cxtau) );
c = AtmospherDensityScaleFactor*VehicleStatus.Drag;

%clamp between 0 and c for numerical safety
c = min(max(c,0),VehicleStatus.Drag);

%simple Euler integration
acc = (-c*oldvel- Params.m*Params.g+VehicleStatus.Thrust)/Params.m + Params.SigmaQ*randn(1);
newvel = oldvel+acc*dT;
newpos = oldpos+oldvel*dT+0.5*acc*dT^2;
XTrue(:,k) = [newpos;newvel];



%---------------- LANDER CONTROL --------------------------%

function DoControl(k,XEst)

global Params;global VehicleStatus;

if(XEst(1)<Params.OpenChuteHeight & ~VehicleStatus.ChuteOpen)
    %open parachute:
    VehicleStatus.ChuteOpen = 1;
    VehicleStatus.Drag = Params.ChuteDrag;
    fprintf('Opening Chute at time %f\n',k*Params.dT);
    Params.ChuteTime = k*Params.dT;
end;

if(XEst(1)<Params.RocketBurnHeight )
    if(~VehicleStatus.RocketsOn)
        fprintf('Releasing Chute at time %f\n',k*Params.dT);
        fprintf('Firing Rockets at time %f\n',k*Params.dT);
        Params.BurnTime = k*Params.dT;
    end;

    %turn on thrusters
    VehicleStatus.RocketsOn = 1;
```

```
    %drop chute..
    VehicleStatus.Drag = 0;

    %simple littel  controller  here (from v^2 = u^2+2as) and +mg for weight of vehicle
    VehicleStatus.Thrust = (Params.m*XEst(2)^2-1)/(2*XEst(1))+0.99*Params.m*Params.g;

end;

if(XEst(1)<1)
    %stop when we hit the ground...
    fprintf('Landed at time %f\n',k*Params.dT);
    VehicleStatus.Landed = 1;
    Params.LandingTime = k*Params.dT;
    break;
end;

return;

%---------------- MANAGE RESULTS STORAGE ------------------------%
function DoStore(k,XEst,PEst,Innovation,S,z)
global Store;
if(k==1)
    Store.XEst  = XEst;
    Store.PEst  = diag(PEst);
    Store.Innovation = Innovation;
    Store.S = S;
    Store.z = z;

else
    Store.XEst = [Store.XEst XEst];
    Store.PEst  = [Store.PEst diag(PEst)];
    Store.Innovation = [ Store.Innovation Innovation];
    Store.S = [Store.S diag(S)];
    Store.z = [Store.z  z];

end;
return;
```

11.2 Matlab Code For Ackerman Model Example

```
function AckermannPredict
clear all;
close all;

dT = 0.1;%time steps size
nSteps = 600;%length of run
L = 2;%length of vehicle
SigmaV = 0.1; %3cm/s std on speed
SigmaPhi = 4*pi/180; % steer inaccuracy

%initial  knowledge pdf (prior @ k = 0)
P = diag ([0.2,0.2,0]); x  = [0;0;0]; xtrue = x;

Q = diag([SigmaV^2 SigmaPhi^2]);
```

```
%--------- Set up graphics ------------%
figure(1);hold on;axis equal;grid on;axis([-20 20 -5 25])

xlabel('x');ylabel('y');
title('uncertainty bounds for Ackermann model');

%--------- Main loop ------------%
for(k = 1:nSteps)
    %control is a wiggle at constant velocity
    u = [1;pi/5*sin(4*pi*k/nSteps)];

    %calculate jacobians
    JacFx = [1 0 -dT*u(1)*sin(x(3)); 0 1 dT*u(1)*cos(x(3)); 0 0 1];
    JacFu = [dT*cos(x(3)) 0; dT*sin(x(3)) 0; dT* tan(u(2))/L dT *u(1)*sec(u(2))^2];

    %prediction steps
    P = JacFx * P *JacFx' + JacFu*Q*JacFu';
    xtrue = AckermannModel(xtrue,u+[SigmaV ;SigmaPhi].*randn(2,1),dT,L);
    x = AckermannModel(x,u,dT,L);


    %draw occasionally
    if(mod(k-1,30)==0)
        PlotEllipse(x,P,0.5); DrawRobot(x,'r');plot(xtrue(1),xtrue(2),'ko');
    end;
end;

print -deps 'AckermannPredict.eps'%save the picture

%------------- MODEL ---------------%
function y = AckermannModel(x,u,dT,L)
y(1,1) = x(1) + dT*u(1)*cos(x(3));
y(2,1) = x(2) + dT*u(1)*sin(x(3));
y(3,1) = x(3) + dT*u(1)/L*tan(u(2));

%--------- Drawing Covariance -----%
function eH = PlotEllipse(x,P,nSigma)
P = P(1:2,1:2); % only plot x-y part
x = x(1:2);
if(~any(diag(P)==0))
    [V,D] = eig(P);
    y = nSigma*[cos(0:0.1:2*pi);sin(0:0.1:2*pi)];
    el = V*sqrtm(D)*y;
    el = [el el(:,1)]+repmat(x,1,size(el,2)+1);
    eH = line(el(1,:), el(2,:));
end;

%--------- Drawing Vehicle -----%
function DrawRobot(Xr,col);

p=0.02; % percentage of axes size
a=axis;
l1=(a(2)-a(1))*p;
l2=(a(4)-a(3))*p;
```

```
P=[−1 1 0 −1; −1 −1 3 −1];%basic triangle
theta = Xr(3)−pi/2;%rotate to point along x axis (theta = 0)
c=cos(theta);
s=sin(theta);
P=[c −s; s c]∗P; %rotate by theta
P(1,:)=P(1,:)∗l1+Xr(1); %scale and shift to x
P(2,:)=P(2,:)∗l2+Xr(2);
H = plot(P(1,:),P(2,:), col, 'LineWidth',0.1);% draw
plot(Xr(1),Xr(2),sprintf('%s+',col));
```

11.3 Matlab Code For EKF Localisation Example

```
function EKFLocalisation
close all; clear all;
global xTrue;global Map;global RTrue;global UTrue;global nSteps;

nSteps = 6000;

Map = 140∗rand(2,30)−70;

UTrue = diag([0.01,0.01,1∗pi/180]).^2;
RTrue = diag([2.0,3∗pi/180]).^2;

UEst = 1.0∗UTrue;
REst = 1.0∗RTrue;

xTrue = [1;−40;−pi/2];
xOdomLast = GetOdometry(1);

%initial conditions:
xEst =xTrue;
PEst = diag([1,1,(1∗pi/180)^2]);

%%%%%%%%% storage %%%%%%%%
InnovStore = NaN∗zeros(2,nSteps);
SStore = NaN∗zeros(2,nSteps);
PStore = NaN∗zeros(3,nSteps);
XStore = NaN∗zeros(3,nSteps);
XErrStore = NaN∗zeros(3,nSteps);

%initial graphics
figure(1); hold on; grid off; axis equal;
plot(Map(1,:),Map(2,:),'g∗');hold on;
set(gcf,'doublebuffer','on');
hObsLine = line([0,0],[0,0]);
set(hObsLine,'linestyle',':');

for k = 2:nSteps

    %do world iteration
    SimulateWorld(k);

    %figure out control
    xOdomNow = GetOdometry(k);
```

```
u = tcomp(tinv(xOdomLast),xOdomNow);
xOdomLast = xOdomNow;

%do prediction
xPred = tcomp(xEst,u);
xPred(3) = AngleWrap(xPred(3));
PPred = J1(xEst,u)* PEst *J1(xEst,u)' + J2(xEst,u)* UEst * J2(xEst,u)';

%observe a randomn feature
[z,iFeature] = GetObservation(k);

if(~isempty(z))
    %predict observation
    zPred = DoObservationModel(xPred,iFeature,Map);

    % get observation Jacobian
    jH = GetObsJac(xPred,iFeature,Map);

    %do Kalman update:
    Innov = z-zPred;
    Innov(2) = AngleWrap(Innov(2));

    S = jH*PPred*jH'+REst;
    W = PPred*jH'*inv(S);
    xEst = xPred+ W*Innov;
    xEst(3) = AngleWrap(xEst(3));

    %note use of 'Joseph' form which is numerically stable
    I = eye(3);
    PEst = (I-W*jH)*PPred*(I-W*jH)'+ W*REst*W';
    PEst = 0.5*(PEst+PEst');

else
    %Ther was no observation available
    xEst = xPred;
    PEst = PPred;
    Innov = [NaN;NaN];
    S = NaN*eye(2);
end;

if(mod(k-2,300)==0)
    DoVehicleGraphics(xEst,PEst (1:2,1:2),8,[0,1]);
    if(~isempty(z))
        set(hObsLine,'XData',[xEst(1),Map(1,iFeature)]);
        set(hObsLine,'YData',[xEst(2),Map(2,iFeature)]);
    end;
    drawnow;
end;

%store results:
InnovStore(:,k) = Innov;
PStore(:,k) = sqrt(diag(PEst));
SStore(:,k) = sqrt(diag(S));
XStore(:,k) = xEst;
XErrStore(:,k) = xTrue-xEst;
```

```
end;

DoGraphs(InnovStore,PStore,SStore,XStore,XErrStore);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function DoGraphs(InnovStore,PStore,SStore,XStore,XErrStore)

figure(1); print -depsc 'EKFLocation.eps'

figure(2);
subplot(2,1,1);plot(InnovStore(1,:));hold on;plot(SStore(1,:),'r');plot(-SStore(1,:),'r')
title('Innovation');ylabel('range');
subplot(2,1,2);plot(InnovStore(2,:)*180/pi);hold on;plot(SStore(2,:)*180/pi,'r');plot(-SStore(2,:)*180/pi,'r')
ylabel('Bearing (deg)');xlabel('time');
print -depsc 'EKFLocationInnov.eps'

figure(2);
subplot(3,1,1);plot(XErrStore(1,:));hold on;plot(3*PStore(1,:),'r');plot(-3*PStore(1,:),'r');
title('Covariance and Error');ylabel('x');
subplot(3,1,2);plot(XErrStore(2,:));hold on;plot(3*PStore(2,:),'r');plot(-3*PStore(2,:),'r')
ylabel('y');
subplot(3,1,3);plot(XErrStore(3,:)*180/pi);hold on;plot(3*PStore(3,:)*180/pi,'r');plot(-3*PStore(3,:)*180/pi,'r')
ylabel('\theta');xlabel('time');
print -depsc 'EKFLocationErr.eps'

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [z,iFeature] = GetObservation(k)
global Map;global xTrue;global RTrue;global nSteps;

%fake sensor failure here
if(abs(k-nSteps/2)<0.1*nSteps)
    z = [];
    iFeature = -1;
else
    iFeature = ceil(size(Map,2)*rand(1));
    z = DoObservationModel(xTrue, iFeature,Map)+sqrt(RTrue)*randn(2,1);
    z(2) = AngleWrap(z(2));
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [z] = DoObservationModel(xVeh, iFeature,Map)
Delta = Map(1:2,iFeature)-xVeh(1:2);
z = [norm(Delta);
    atan2(Delta(2),Delta(1))-xVeh(3)];
z(2) = AngleWrap(z(2));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function SimulateWorld(k)
global xTrue;
u = GetRobotControl(k);
xTrue = tcomp(xTrue,u);
xTrue(3) = AngleWrap(xTrue(3));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function jH = GetObsJac(xPred, iFeature,Map)
jH = zeros(2,3);
Delta = (Map(1:2,iFeature)-xPred(1:2));
r = norm(Delta);
jH(1,1) = -Delta(1) / r;
jH(1,2) = -Delta(2) / r;
jH(2,1) = Delta(2) / (r^2);
jH(2,2) = -Delta(1) / (r^2);
jH(2,3) = -1;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xnow] = GetOdometry(k)
 persistent  LastOdom; %internal to robot low-level controller
global UTrue;
 if(isempty(LastOdom))
     global xTrue;
     LastOdom = xTrue;
end;
u = GetRobotControl(k);
xnow =tcomp(LastOdom,u);
uNoise = sqrt(UTrue)*randn(3,1);
xnow = tcomp(xnow,uNoise);
LastOdom = xnow;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function u = GetRobotControl(k)
global nSteps;
u  = [0; 0.025 ; 0.1* pi/180*sin(3*pi*k/nSteps)];
%u  = [0; 0.15 ; 0.3* pi/180];
```

11.4 Matlab Code For EKF Mapping Example

```
function EKFLocalisation
close all; clear all;
global xVehicleTrue;global Map;global RTrue;global UTrue;global nSteps;


nSteps = 600;
nFeatures = 6;
MapSize = 200;
Map = MapSize*rand(2,nFeatures)-MapSize/2;

UTrue = diag([0.01,0.01,1*pi/180]).^2;
RTrue = diag([8.0,7*pi/180]).^2;

UEst = 1.0*UTrue;
REst = 1.0*RTrue;

xVehicleTrue = [1;-40;-pi/2];

%initial  conditions - no map:
xEst =[];
PEst = [];
MappedFeatures = NaN*zeros(nFeatures,2);
```

```
%storage:
PStore = NaN*zeros(nFeatures,nSteps);
XErrStore = NaN*zeros(nFeatures,nSteps);


%initial graphics - plot true map
figure(1); hold on; grid off; axis equal;
plot(Map(1,:),Map(2,:),'g*');hold on;
set(gcf,'doublebuffer','on');
hObsLine = line ([0,0],[0,0]);
set(hObsLine,'linestyle',':');



for k = 2:nSteps

    %do world iteration
    SimulateWorld(k);

    %simple prediction model:
    xPred = xEst;
    PPred = PEst;

    %observe a randomn feature
    [z,iFeature] = GetObservation(k);

    if(~isempty(z))

        %have we seen this feature before?
        if( ~isnan(MappedFeatures(iFeature,1)))

            %predict observation: find out where it is in state vector
            FeatureIndex = MappedFeatures(iFeature,1);
            xFeature = xPred(FeatureIndex:FeatureIndex+1);
            zPred = DoObservationModel(xVehicleTrue,xFeature);

            % get observation Jacobians
            [jHxv,jHxf] = GetObsJacs(xVehicleTrue,xFeature);

            % fill in state jacobian
            jH = zeros(2,length(xEst));
            jH(:,FeatureIndex:FeatureIndex+1) = jHxf;

            %do Kalman update:
            Innov = z-zPred;
            Innov(2) = AngleWrap(Innov(2));

            S = jH*PPred*jH'+REst;
            W = PPred*jH'*inv(S);
            xEst = xPred+ W*Innov;

            PEst = PPred-W*S*W';
            %note use of 'Joseph' form which is numerically stable
%            I = eye(size(PEst));
```

```
%            PEst = (I-W*jH)*PPred*(I-W*jH)'+ W*REst*W';

            %ensure P remains symmetric
            PEst = 0.5*(PEst+PEst');
        else
            % this is a new feature add it to the map....
            nStates = length(xEst);

            xFeature = xVehicleTrue(1:2)+ [z(1)*cos(z(2)+xVehicleTrue(3));z(1)*sin(z(2)+xVehicleTrue(3))];
            xEst = [xEst;xFeature];
            [jGxv, jGz] = GetNewFeatureJacs(xVehicleTrue,z);

            M = [eye(nStates), zeros(nStates,2);% note we don't use jacobian w.r.t vehicle
                zeros(2,nStates)  , jGz];

            PEst = M*blkdiag(PEst,REst)*M';

            %remember this feature as being mapped we store its ID and position in the state vector
            MappedFeatures(iFeature,:) = [length(xEst)-1, length(xEst)];

        end;

    else
        %There was no observation available

    end;

    if(mod(k-2,40)==0)
        plot(xVehicleTrue(1),xVehicleTrue(2),'r*');

        %now draw all the estimated feature points
        DoMapGraphics(xEst,PEst,5);

        fprintf('k = %d\n',k);

        drawnow;
    end;


    %Storage:
    for(i = 1:nFeatures)
        if(~isnan(MappedFeatures(i,1)))
            iL =MappedFeatures(i,1);
            PStore(k,i) = det(PEst(iL:iL+1,iL:iL+1));
            XErrStore(k,i) = norm(xEst(iL:iL+1)-Map(:,i));
        end;
    end;


end;


figure(2);
plot(PStore);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [z,iFeature] = GetObservation(k)
global Map;global xVehicleTrue;global RTrue;global nSteps;

%choose a random feature to see from True Map
iFeature = ceil(size(Map,2)*rand(1));
z = DoObservationModel(xVehicleTrue,Map(:,iFeature))+sqrt(RTrue)*randn(2,1);
z(2) = AngleWrap(z(2));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [z] = DoObservationModel(xVeh, xFeature)
Delta = xFeature-xVeh(1:2);
z = [norm(Delta);
    atan2(Delta(2),Delta(1))-xVeh(3)];
z(2) = AngleWrap(z(2));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function SimulateWorld(k)
global xVehicleTrue;
u = GetRobotControl(k);
xVehicleTrue = tcomp(xVehicleTrue,u);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [jHxv,jHxf] = GetObsJacs(xPred, xFeature)
jHxv = zeros(2,3);jHxf = zeros(2,2);
Delta = (xFeature-xPred(1:2));
r = norm(Delta);
jHxv(1,1) = -Delta(1) / r;
jHxv(1,2) = -Delta(2) / r;
jHxv(2,1) = Delta(2) / (r^2);
jHxv(2,2) = -Delta(1) / (r^2);
jHxv(2,3) = -1;
jHxf(1:2,1:2) = -jHxv(1:2,1:2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [jGx,jGz] = GetNewFeatureJacs(Xv, z);
x = Xv(1,1);
y = Xv(2,1);
theta = Xv(3,1);
r = z(1);
bearing = z(2);
jGx = [ 1   0   -r*sin(theta + bearing);
        0   1   r*cos(theta + bearing)];
jGz = [ cos(theta + bearing) -r*sin(theta + bearing);
        sin(theta + bearing) r*cos(theta + bearing)];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function u = GetRobotControl(k)
global nSteps;
%u = [0; 0.25 ; 0.3* pi/180*sin(3*pi*k/nSteps)];
u  = [0; 0.15 ; 0.3* pi/180];
```

11.5 Matlab Code For EKF SLAM Example

```
function EKFSLAM
close all; clear all;
global xVehicleTrue;global Map;global RTrue;global UTrue;global nSteps;
global SensorSettings;

%change these to alter sensor behaviour
SensorSettings.FieldOfView = 45;
SensorSettings.Range = 100;

%how often shall we draw?
DrawEveryNFrames = 50;

%length of experiment
nSteps = 8000;


%when to take pictures?
SnapShots = ceil(linspace(2,nSteps,25));


%size of problem
nFeatures = 40;
MapSize = 200;
Map = MapSize*rand(2,nFeatures)-MapSize/2;

UTrue = diag([0.01,0.01,1.5*pi/180]).^2;
RTrue = diag([1.1,5*pi/180]).^2;

UEst = 2.0*UTrue;
REst = 2.0*RTrue;

xVehicleTrue = [0;0;-pi/2];

%initial conditions - no map:
xEst =[xVehicleTrue];
PEst = diag([1,1,0.01]);
MappedFeatures = NaN*zeros(nFeatures,2);

%storage:
PStore = NaN*zeros(nFeatures,nSteps);
XErrStore = NaN*zeros(nFeatures,nSteps);


%initial graphics - plot true map
figure(1); hold on; grid off; axis equal;
plot(Map(1,:),Map(2,:),'g*');hold on;
set(gcf,'doublebuffer','on');
hObsLine = line ([0,0],[0,0]);
set(hObsLine,'linestyle',':');
a = axis; axis(a*1.1);


xOdomLast = GetOdometry(1);
```

```
for k = 2:nSteps

    %do world iteration
    SimulateWorld(k);

    %figure out control
    xOdomNow = GetOdometry(k);
    u = tcomp(tinv(xOdomLast),xOdomNow);
    xOdomLast = xOdomNow;

    %we'll need this lots ...
    xVehicle = xEst(1:3);
    xMap = xEst(4:end);

    %do prediction (the following is simply the result of multiplying
    %out block form of jacobians)
    xVehiclePred = tcomp(xVehicle,u);
    PPredvv = J1(xVehicle,u)* PEst(1:3,1:3) *J1(xVehicle,u)' + J2(xVehicle,u)* UEst * J2(xVehicle,u)';
    PPredvm = J1(xVehicle,u)*PEst(1:3,4:end);
    PPredmm = PEst(4:end,4:end);

    xPred = [xVehiclePred;xMap];
    PPred = [PPredvv PPredvm;
        PPredvm' PPredmm];

    %observe a randomn feature
    [z,iFeature] = GetObservation(k);

    if(~isempty(z))
        %have we seen this feature before?
        if( ~isnan(MappedFeatures(iFeature,1)))

            %predict observation: find out where it is in state vector
            FeatureIndex = MappedFeatures(iFeature,1);
            xFeature = xPred(FeatureIndex:FeatureIndex+1);

            zPred = DoObservationModel(xVehicle,xFeature);

            % get observation Jacobians
            [jHxv,jHxf] = GetObsJacs(xVehicle,xFeature);

            % fill in state jacobian
            jH = zeros(2,length(xEst));
            jH(:,FeatureIndex:FeatureIndex+1) = jHxf;
            jH(:,1:3) = jHxv;

            %do Kalman update:
            Innov = z-zPred;
            Innov(2) = AngleWrap(Innov(2));

            S = jH*PPred*jH'+REst;
            W = PPred*jH'*inv(S);
            xEst = xPred+ W*Innov;
```

```
            PEst = PPred−W*S*W';

            %ensure P remains symmetric
            PEst = 0.5*(PEst+PEst');
        else
            % this is a new feature add it to the map....
            nStates = length(xEst);

            xFeature = xVehicle(1:2)+ [z(1)*cos(z(2)+xVehicle(3));z(1)*sin(z(2)+xVehicle(3))];
            xEst = [xEst;xFeature]; %augmenting state vector
            [jGxv, jGz] = GetNewFeatureJacs(xVehicle,z);

            M = [eye(nStates), zeros(nStates,2);% note we don't use jacobian w.r.t vehicle
                jGxv zeros(2,nStates−3) , jGz];

            PEst = M*blkdiag(PEst,REst)*M';

            %remember this feature as being mapped we store its ID and position in the state vector
            MappedFeatures(iFeature,:) = [length(xEst)−1, length(xEst)];

        end;
    else
        xEst = xPred;
        PESt = PPred;
    end;


    if(mod(k−2,DrawEveryNFrames)==0)
        a = axis;
        clf;
        axis(a);hold on;
        n  = length(xEst);
        nF = (n−3)/2;
        DoVehicleGraphics(xEst(1:3),PEst (1:3,1:3),3,[0 1]);

        if(~isnan(z))
            h = line([xEst(1),xFeature (1)],[ xEst(2),xFeature (2)]);
            set(h,' linestyle ',' :' );
        end;
        for(i = 1:nF)
            iF = 3+2*i−1;
            plot(xEst(iF),xEst(iF+1),'b*');
            PlotEllipse (xEst(iF:iF+1),PEst(iF:iF+1,iF:iF+1),3);
        end;
        fprintf('k␣=␣%d\n',k);
        drawnow;
    end;

    if(ismember(k,SnapShots))
        iPic = find(SnapShots==k);
        print(gcf,'−depsc',sprintf('EKFSLAM%d.eps',iPic));
    end;


end;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [z,iFeature] = GetObservation(k)
global Map;global xVehicleTrue;global RTrue;global nSteps;global SensorSettings
done = 0;
Trys = 1;
z =[]; iFeature = -1;
while(~done & Trys <0.5*size(Map,2))

    %choose a random feature to see from True Map
    iFeature = ceil(size(Map,2)*rand(1));
    z = DoObservationModel(xVehicleTrue,Map(:,iFeature))+sqrt(RTrue)*randn(2,1);
    z(2) = AngleWrap(z(2));
    %look forward...and only up to 40m
    if(abs(pi/2-z(2))<SensorSettings.FieldOfView*pi/180 & z(1) < SensorSettings.Range)
        done =1 ;
    else
        Trys =Trys+1;
        z =[]; iFeature = -1;
    end;
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [z] = DoObservationModel(xVeh, xFeature)
Delta = xFeature-xVeh(1:2);
z = [norm(Delta);
    atan2(Delta(2),Delta(1))-xVeh(3)];
z(2) = AngleWrap(z(2));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function SimulateWorld(k)
global xVehicleTrue;
u = GetRobotControl(k);
xVehicleTrue = tcomp(xVehicleTrue,u);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [jHxv,jHxf] = GetObsJacs(xPred, xFeature)
jHxv = zeros(2,3);jHxf = zeros(2,2);
Delta = (xFeature-xPred(1:2));
r = norm(Delta);
jHxv(1,1) = -Delta(1) / r;
jHxv(1,2) = -Delta(2) / r;
jHxv(2,1) = Delta(2) / (r^2);
jHxv(2,2) = -Delta(1) / (r^2);
jHxv(2,3) = -1;
jHxf(1:2,1:2) = -jHxv(1:2,1:2);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [jGx,jGz] = GetNewFeatureJacs(Xv, z);
x = Xv(1,1);
y = Xv(2,1);
theta = Xv(3,1);
r = z(1);
bearing = z(2);
jGx = [ 1   0   -r*sin(theta + bearing);
```

```
    0    1    r*cos(theta + bearing)];
jGz = [ cos(theta + bearing) -r*sin(theta + bearing);
    sin(theta + bearing) r*cos(theta + bearing)];


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xnow] = GetOdometry(k)
persistent LastOdom; %internal to robot low-level controller
global UTrue;
if(isempty(LastOdom))
    global xVehicleTrue;
    LastOdom = xVehicleTrue;
end;
u = GetRobotControl(k);
xnow =tcomp(LastOdom,u);
uNoise = sqrt(UTrue)*randn(3,1);
xnow = tcomp(xnow,uNoise);
LastOdom = xnow;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function u = GetRobotControl(k)
global nSteps;
%u = [0; 0.25 ; 0.3* pi/180*sin(3*pi*k/nSteps)];
u = [0; 0.15 ; 0.2* pi/180];

%-------- Drawing Covariance -----%
function eH = PlotEllipse(x,P,nSigma)
eH = [];
P = P(1:2,1:2); % only plot x-y part
x = x(1:2);
if(~any(diag(P)==0))
    [V,D] = eig(P);
    y = nSigma*[cos(0:0.1:2*pi);sin(0:0.1:2*pi)];
    el = V*sqrtm(D)*y;
    el = [el el(:,1)]+repmat(x,1,size(el,2)+1);
    eH = line(el(1,:), el(2,:) );
end;
```

11.6 Matlab Code For Particle Filter Example

```
%Monte-carlo based localisation
%note this is not coded efficiently but rather to make the ideas clear
%all loops should be vectorized but that gets a little matlab-speak intensive
%and may obliterate the elegance of a particle filter ....

function MCL
close all; clear all;
global xTrue;global Map;global RTrue;global UTrue;global nSteps;


nSteps = 6000;

%change this to see how sensitive we are to the number of particle
%(hypotheses run) especially in relation to initial distribution!
```

```
nParticles = 400;

Map = 140*rand(2,30)-70;

UTrue = diag([0.01,0.01,1*pi/180]).^2;
RTrue = diag([2.0,3*pi/180]).^2;

UEst = 1.0*UTrue;
REst = 1.0*RTrue;

xTrue = [1;-40;-pi/2];
xOdomLast = GetOdometry(1);

%initial conditions: - a point cloud around truth
xP =repmat(xTrue,1,nParticles)+diag([8,8,0.4])*randn(3,nParticles);

%%%%%%%%% storage %%%%%%%%

%initial graphics
figure(1); hold on; grid off; axis equal;
plot(Map(1,:),Map(2,:),'g*');hold on;
set(gcf,'doublebuffer','on');
hObsLine = line ([0,0],[0,0]);
set(hObsLine,'linestyle',':');
hPoints = plot(xP(1,:),xP(2,:),'.');

for k = 2:nSteps

    %do world iteration
    SimulateWorld(k);


    %all particles are equally important
    L = ones(nParticles,1)/nParticles;


    %figure out control
    xOdomNow = GetOdometry(k);
    u = tcomp(tinv(xOdomLast),xOdomNow);
    xOdomLast = xOdomNow;

    %do prediction
    %for each particle we add in control vector AND noise
    %the control noise adds diversity within the generation
    for(p = 1:nParticles)
        xP(:,p) = tcomp(xP(:,p),u+sqrt(UEst)*randn(3,1));
    end;

    xP(3,:) = AngleWrap(xP(3,:));


    %observe a randomn feature
    [z,iFeature] = GetObservation(k);

    if(~isempty(z))
```

```
        %predict observation

        for(p = 1:nParticles)
            %what do we expect observation to be for this particle?
            zPred = DoObservationModel(xP(:,p),iFeature,Map);

            %how different
            Innov = z−zPred;

            %get likelihood (new importance). Assume gaussian here but any pdf works!
            %if predicted obs is very different from actual obs this score will be low
            %−>this particle is not very good at representing state. A lower score means
            %it is less likely to be selected for the next generation...
            L(p) = exp(−0.5∗Innov'∗inv(REst)∗Innov)+0.001;

        end;
    end;


    %reselect based on weights:
    %particles with big weights will occupy a greater percentage of the
    %y axis in a cummulative plot
    CDF = cumsum(L)/sum(L);
    %so randomly (uniform) choosing y values is more likely to correspond to
    %more likely (better) particles ...
    iSelect  = rand(nParticles,1);
    %find the particle that corresponds to each y value (just a look up)
    iNextGeneration = interp1(CDF,1:nParticles,iSelect,'nearest','extrap');
    %copy selected particles for next generation..
    xP = xP(:,iNextGeneration);

    %our estimate is simply the mean of teh particles
    xEst = mean(xP,2);

    if(mod(k−2,10)==0)

        figure(1);
        set(hPoints,'XData',xP(1,:));
        set(hPoints,'YData',xP(2,:));
        if(~isempty(z))
            set(hObsLine,'XData',[xEst(1),Map(1,iFeature)]);
            set(hObsLine,'YData',[xEst(2),Map(2,iFeature)]);
        end;
        figure(2);plot(xP(1,:),xP(2,:),'.');
        drawnow;
    end;

end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [z,iFeature] = GetObservation(k)
global Map;global xTrue;global RTrue;global nSteps;

%fake sensor failure here
if(abs(k−nSteps/2)<0.1∗nSteps)
```

```
    z = [];
    iFeature = −1;
else
    iFeature = ceil(size(Map,2)∗rand(1));
    z = DoObservationModel(xTrue, iFeature,Map)+sqrt(RTrue)∗randn(2,1);
    z(2) = AngleWrap(z(2));
end;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [z] = DoObservationModel(xVeh, iFeature,Map)
Delta = Map(1:2,iFeature)−xVeh(1:2);
z = [norm(Delta);
    atan2(Delta(2),Delta(1))−xVeh(3)];
z(2) = AngleWrap(z(2));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function SimulateWorld(k)
global xTrue;
u = GetRobotControl(k);
xTrue = tcomp(xTrue,u);
xTrue(3) = AngleWrap(xTrue(3));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [xnow] = GetOdometry(k)
persistent LastOdom; %internal to robot low−level controller
global UTrue;
if(isempty(LastOdom))
    global xTrue;
    LastOdom = xTrue;
end;
u = GetRobotControl(k);
xnow =tcomp(LastOdom,u);
uNoise = sqrt(UTrue)∗randn(3,1);
xnow = tcomp(xnow,uNoise);
LastOdom = xnow;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function u = GetRobotControl(k)
global nSteps;
u = [0; 0.025 ; 0.1∗ pi/180∗sin(3∗pi∗k/nSteps)];
```